WYŻSZA SZKOŁA INFORMATYKI STOSOWANEJ I ZARZĄDZANIA
pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

STUDIA INŻYNIERSKIE



PRACA DYPLOMOWA

Jacek Wolski

Solvery SAT - algorytmy i ich działanie

Praca wykonana pod kierunkiem:

dr Piotr Sapiecha

Spis treści

Spis Algorytmów

Spis Ilustracji

wartościowanie, powoduje, że dane wartościowanie w losowy, heurystyczny sposób jest poprawiane. Nowa wersja wartości zmiennych jest weryfikowana. Czynności te są powtarzane, dopóki nie zostanie znalezione wartościowanie spełniające bądź nie

Spis Tabel

1 Wstęp

We współczesnym świecie pomimo rozwoju tak wielu gałęzi nauk szczegółowych istnieją jeszcze dziedziny badawcze, w których ich podstawowe pytania pozostają nierozstrzygnięte. Jedną z takich dziedzin jest teoria algorytmów. Wśród wielu problemów, które są w niej rozpatrywane, kluczowym jest problem spełnialności formuł logicznych. Problem ten zawiera w sobie nie tylko elementy teoretyczne, ale także rzutuje na zagadnienia praktyczne. Dzięki temu ciągle podejmowane są badania nad tą częścią nauk szczegółowych.

W niniejszej pracy zostaną przedstawione zarówno klasyczne jak i nowoczesne algorytmy rozwiązywania problemu spełnialności formuł logicznych. Ponadto zostaną zaprezentowane nowoczesne aplikacje, które służą do badania spełnialności tzw. solvery SAT. Te aplikacje zostaną przedstawione w ujęciu ogólnym w efekcie analizy ich działania. W końcowej części tej pracy zostaną pokazane wyniki testów tych solverów.

2 Cele Pracy

Niniejsza praca omawia klasyczne algorytmy rozwiązywania problemu spełnialności formuł logicznych. Problem spełnialności formuł logicznych (SAT) jest jednym z głównych problemów w współczesnej teorii złożoności obliczeniowej.

Cele niniejszej pracy zawierają się w następujących punktach :

1. Przedstawienie algorytmów rozwiązywania problemu SAT
2. Przedstawienie najnowszych rozwiązań w tej dziedzinie i współczesnych osiągnięć
3. Analiza działania wybranych, współczesnych solverów SAT

3 Prezentacja SAT

3.1 Rys historyczny

Korzenie teorii NP-zupełności (NP-completeness) sięgają lat trzydziestych ubiegłego wieku. Pojawiły się wówczas w pracach takich naukowców jak K. Goedel czy też A. Turing pojęcia dotyczące języków oraz ich rozpoznawania. W latach 60-tych w trakcie rozwijania teorii złożoności obliczeniowej wprowadzono pojęcie algorytmu wielomianowego (polynomial algorithm).

W 1971 r. po raz pierwszy Stephen Cook wprowadził pojęcie „problemu NP-zupełnego" (NP-complete problem) oraz dowiódł, że SAT oraz 3-SAT należą do zbioru takich problemów [4].

3.2 Twierdzenie Cooka

Zgodnie z twierdzeniem Cooka, problem SAT jest NP-zupełny (NP-complete). Oznacza to, że :

- problem jest NP;
- każdy inny problem NP-zupełny jest sprowadzalny do tego problemu w czasie wielomianowym;

Jeżeli problem należy do klasy NP, to oznacza, że niedeterministyczna maszyna Turinga przedstawi rozwiązanie w czasie co najwyżej wielomianowym. To twierdzenie jest równoważne z twierdzeniem, że deterministyczna maszyna Turinga może rozwiązać zagadnienie w czasie wykładniczym. W rezultacie nie jest znany żaden algorytm dla takiego problemu, który działałby w czasie wielomianowym.

Jeżeli problem NP-zupełny P_1 jest sprowadzalny do hipotetycznego problemu P_2 w czasie wielomianowym, to znaczy, że można dokonać przekształcenia tego problemu P_1 przy pomocy odpowiedniej funkcji w czasie wielomianowym w problem P_2 w taki sposób, że dla każdego przykładu problemu P_1 wynik będzie taki sam jak dla przykładu po transformacji. Ponadto złożoność obliczeniowa takiego przekształconego problemu nie będzie większa niż przed tą operacją [5],[3].

Powyżej posługuję się pojęciami „czas wielomianowy" i „czas wykładniczy". Te pojęcia są związane z dwoma, różnymi zagadnieniami. Pierwsze z nich dotyczy złożoności obliczeniowej algorytmu czyli takiej funkcji, która przyporządkowuje każdej wartości rozmiaru konkretnego problemu maksymalną ilość kroków elementarnych (lub jednostek czasu pracy). W ten sposób można dokonać porównania różnych algorytmów. W tym ujęciu „algorytm o czasie wielomianowym" oznacza złożoność tego algorytmu rzędu co najmniej $O(n^k)$, gdzie n zależy od zmiennych problemu, a k to liczba naturalna. Podobnie „algorytm o czasie wykładniczym" oznacza złożoność rzędu co najmniej $O(2^n)$, gdzie n jest zależna od zmiennych problemu.

Drugie zagadnienie jest związane z rzędem czasu potrzebnego do wykonania określonych operacji. Dane przekształcenie działa w czasie wielomianowym, jeżeli ten czas można opisać wielomianem rzędu co najmniej n^k, gdzie podobnie jak powyżej należy zinterpretować znaczenie n i k. Analogiczna interpretacja występuje w przypadku czasu wykładniczego [3].

Poniżej przedstawiam zależność pomiędzy n, a różnymi rzędami wielkości w celu pokazania różnic, jakie występują pomiędzy czasem wielomianowym a wykładniczym. (przy założeniu, że elementarny krok trwa 1 μs).

Rząd wielkości	n=10	n=60
n	0,00001 s	0,00006 s
n^5	0,1 s	13 min
2^n	0,001 s	3366 wieków
3^n	0,059 s	$1,3 \times 10^{13}$ wieków

Tabela 1 Zależność między czasem wielomianowym a czasem wykładniczym

Problemy oraz ich algorytmy są dzielone na dwie klasy w zależności od rzędu złożoności obliczeniowej. Część problemów jest klasy P. Natomiast wiele problemów jest NP. Dotychczas rozpoznano kilka tysięcy takich problemów z różnych dziedzin nauki na przykład problem komiwojażera, wspomniany już SAT, ILP (Integer Linear Programming), problem kliki itd.

Potencjalne rozwiązanie w czasie wielomianowym jednego problemu NP-zupełnego, oznacza możliwość rozwiązania wszystkich zagadnień z tej klasy. Dlatego od kilkunastu lat trwają prace nad tą częścią teorii algorytmiki. W dalszym ciągu są

zakończone niepowodzeniem [5].

Dla pełnego zrozumienia zagadnień związanych z klasami problemów P i NP, które powyżej zostały zarysowane, oraz wspomnianą powyżej maszyną Turinga proponuję pozycję [2].

3.3 Zarys SAT

Podstawą problemu SAT jest formuła logiczna, która składa się z :

- zmiennych logicznych (literałów) : a, b, c ...;
- spójników logicznych czyli takich funkcji logicznych, które mają jeden lub dwa argumenty i jedną wartość wyjściową przykładowo negacja (NOT, $\neg$, ~), alternatywa (OR, $\vee$, +), koniunkcja (AND, $\wedge$, $\cdot$), implikacja ($\Rightarrow$), równoważność ($\Leftrightarrow$);
- nawiasów (klauzul);

Każda formuła logiczna F posiada określone wartościowania czyli zbiory wartości zmiennych. Jeśli istnieje takie przyporządkowanie wartości zmiennych, że przyjmie wartość logiczną równą 1, to wartościowanie takie nazywamy **spełniającym**, a formułę **- spełnialną**. Istnienie lub brak wartościowania spełniającego daną formułę jest istotą problemu SAT (SAT - „satisfiablity" czyli „spełnialność") [5].

Przykładowo niech będzie dana formuła F :

$$F = (\neg a \wedge b) \vee (\neg b \wedge c);$$

co jest równoznaczne z : $F = \sim a \cdot b + \sim b \cdot c;$

Świadomie pominąłem funkcje wynikania i równoważności, ponieważ te funkcje można przedstawić przy pomocy pozostałych spójników logicznych. Przykładowo, $a \Rightarrow b$ można zapisać w postaci $\sim(a \cdot \sim b)$.

Poniżej przedstawiam wartościowania przykładowej formuły wraz z jej wartością:

a	b	c	~a · b	~b · c	F
0	0	0	0	0	0
0	**0**	**1**	**0**	**1**	**1**
0	**1**	**0**	**1**	**0**	**1**
0	**1**	**1**	**1**	**0**	**1**
1	0	0	0	0	0
1	**0**	**1**	**0**	**1**	**1**
1	1	0	0	0	0
1	1	1	0	0	0

Tabela 2 Przykładowe wartościowania formuły

Zgodnie z powyższym, przykładowa formuła posiada osiem wartościowań. Spośród nich cztery są spełniające (zaznaczone). Tak więc formuła jest spełniona dla wartości (a,b,c) = {(001),(010),(011),(101)}

Powyższa tabela pokazuje najprostszy algorytm dla tego problemu. Polega na przejrzeniu kolejno wszystkich wartościowań w całej dziedzinie formuły, przy jednoczesnym obliczaniu wartości badanej formuły. Nie jest to efektywne rozwiązanie, ponieważ dla formuły o n zmiennych istnieje 2^n możliwych wartościowań. Tak więc przedstawiony algorytm „naiwny" dla formuły o liczbie zmiennych n=100 i 1 ns dla każdego przypisania wartości i obliczenia wartości, sprawdzi wszystkie możliwości w ciągu 10^{13} lat [5],[11].

3.4 Rodzaje SAT

Przedstawiony wyżej problem jest wersją ogólną zagadnienia. W zależności od np. formy czy też liczby literałów rozróżnia się następujące odmiany :

- CIRCUIT-SAT Problem:

INPUT: Układ elektroniczny składający się z bramek logicznych AND, OR i NOT ;

OUTPUT: Czy istnieje wartościowanie spełniające ?

Jest to specyficzna wersja wyżej przedstawionego problemu SAT. Istnieje dowód, że CIRCUIT-SAT jest NP-zupełny [5].

- **k-SAT Problem**:

INPUT: Formuła logiczna, która w każdej klauzuli zawiera dokładnie k literałów;

OUTPUT: Czy istnieje wartościowanie spełniające ?

- **HORN-SAT Problem**:

INPUT: Formuła logiczna zawierająca klauzule, które składają się z n literałów. Dokładnie jeden literał jest pozytywny, pozostałe są negatywne (klauzule Horna);

OUTPUT: Czy istnieje wartościowanie spełniające ?

- **MAXSAT Problem** :

INPUT: Formuła logiczna bez żadnych ograniczeń;

OUTPUT: Czy istnieje wartościowanie spełniające, które spełnia co najmniej k klauzul ?

- **NAESAT Problem** :

INPUT: Formuła logiczna bez ograniczeń;

OUTPUT: Czy istnieje wartościowanie spełniające, które w danej klauzuli nie dopuszcza do przyjęcia przez literały tej samej wartości ?

- **CNF-SAT Problem** :

INPUT: Formuła logiczna przedstawiona w koniunkcyjnej postaci normalnej (*conjunctive normal form*).

OUTPUT: Czy istnieje wartościowanie spełniające ?

Konjunktywna postać normalna formuły oznacza, że jest zapisana jako koniunkcja klauzul, przy czym każda klauzula jest alternatywą jednego lub więcej literałów. Przykładem może być formuła : $(a + \sim b) \cdot (b + c + d) \cdot (\sim a + d) \cdot \ldots$.

- DNF-SAT Problem :

INPUT: Formuła logiczna przedstawiona w dysjunktywnej postaci normalnej (*disjunctive normal form*).

OUTPUT: Czy istnieje wartościowanie spełniające ?

Dysjunktywna postać normalna formuły charakteryzuje się tym, że pomiędzy klauzulami istnieje związek alternatywy, a literały są połączone funkcją koniunkcji przykładowo (a · ~b) + (b · d) + (~a · d) + … .

- MV-SAT Problem :

INPUT: Zmienne w formule logicznej mogą przyjmować inne wartości oprócz zwykłych wartości binarnych.

OUTPUT: Czy istnieje wartościowanie spełniające ?

Niektóre problemy z zakresu weryfikacji logicznej są łatwiejsze do sformułowania, jeśli przyjmie się istnienie trzeciej możliwej wartości oznaczającej *„don't care"* (tzn. wartość nie wpływa na całą formułę).

Podana powyżej klasyfikacja nie jest pełna. Istnieją problemy, które powstają poprzez połączenie dwóch z powyżej przedstawionych np. 3-CNF-SAT - połączenie 3-SAT pod względem liczby literałów w klauzuli i CNF-SAT pod względem postaci przedstawiania danej formuły.

W przypadku SAT warty wspomnienia jest fakt, że niektóre wersje SAT nie należą do zbioru problemów NP-zupełnych. Jeśli rozpatrzymy problem k-SAT okaże się, że zarówno dla 1-SAT jak i 2-SAT są znane algorytmy działające w czasie wielomianowym. Z drugiej strony 3-SAT, który można przedstawiać jako uogólnienie 2-SAT (przejście pomiędzy tymi dwoma problemami poprzez powtarzanie literałów), należy do klasy problemów NP-zupełnych [9].

3.5 Zastosowanie SAT

Problem SAT znajduje zastosowanie w wielu dziedzinach związanych z techniką cyfrową. Występuje przy takich zagadnieniach jak ATPG (Automatic Test Pattern Generation) lub BMC (Bounded Model Checking) czy też przy

testowaniu i weryfikacji układów logicznych (FPGA Routing, Path Delay Analysis). Jednocześnie badania prowadzone nad tym problemem dostarczają wyników dla takich działów jak kryptologia i kryptografia ([5],[7]).

Równocześnie badania prowadzone nad tym problemem służą rozwojowi teorii algorytmów i teorii złożoności obliczeniowej.

4 SAT solvery

4.1 Wprowadzenie

Od kilkunastu lat trwają prace nad rozwiązaniem problemu SAT dla formuł pochodzących min. z zagadnień techniki cyfrowej w efektywnym czasie. Oznacza to algorytmy, które można wykorzystać przy zastosowaniach komercyjnych. Badania są prowadzone na całym świecie między innymi na takich uniwersytetach jak Berkeley, Princeton lub MIT. Głównym celem jest doprowadzić do opracowania praktycznego algorytmu, który można zastosować do sprawdzenia spełnialności dużych formuł logicznych rzędu tysięcy, a nawet milionów zmiennych. Aplikacje, które są budowane w wyniku tych prac to solvery SAT czyli programy komputerowe, które są specjalnie przygotowywane i opracowywane pod kątem rozwiązywania problemu spełnialności formuł logicznych [7].

Prace te w ciągu ostatnich kilkunastu lat podlegają ciągłemu rozwojowi, czego dowodem są specjalne zawody *„SAT Contest"*. W 2004 roku odbyły się w Francji. Zostało zgłoszone ponad pięćdziesiąt różnych aplikacji pochodzących od dwudziestu ośmiu różnych twórców z całego świata. Zadaniem proponowanych programów było rozwiązać konkretne zadanie w określonym i akceptowalnym z punktu widzenia człowieka czasie. Przykładowo w trakcie zawodów *„SAT'04 Contest"* każda z proponowanych aplikacji otrzymała na pracę 600 sekund czasu procesora. Badania przeprowadzano na dwóch jednostkach komputerowych : LRI - Athlon XP 1,8 GHz i 1 GB RAM, oraz ß-lab - INTEL Xeon 2,4 GHz i 1 GB RAM. Organizatorzy przygotowali również pakiety specjalnych testów w dziewięciu różnych kategoriach. Przykładowo w kategorii RANDOM przygotowano 30 serii po 10 formuł. Piętnaście serii było sprowadzonych do formatu 3-CNF (konjunktywna postać normalna i tylko trzy literały na klauzule) i zawierało różną ilość zmiennych (od 400 do 900). Następne 15 serii również było przygotowanych w formacie CNF, ale liczba dopuszczalnych literałów w klauzuli *k* zmieniała się od 3 do 9. Liczba zmiennych zależała od liczby *k* [12]. Powyższe informacje świadczą o zainteresowaniu przy rozwiązywaniu tego zagadnienia.

4.2 Rys historyczny

Początki prac nad solverami SAT sięgają lat 60-tych. Dotyczyły na początku zagadnień przeprowadzania dowodów logicznych przy pomocy maszyn obliczeniowych. W 1960 roku M.Davis i H. Putnam opublikowali artykuł, w którym zaproponowali algorytm DP (Davisa-Putnama), który mógł sprawdzać spełnialność niewielkich formuł. W 1962 roku M. Davis, G. Logemann i D. Loveland przedstawili pracę prezentującą algorytm, który jest do dzisiaj ramą dla wielu SAT solverów. Został on nazwany algorytmem DLL (od nazwisk twórców). W 1986 roku R. Bryant zaproponował metodę wykorzystującą binarne diagramy decyzyjne (BDD - binary decision diagrams). Formuła była reprezentowana przez acykliczny graf skierowany. W 1992 roku B. Selman, H. Levesque i D. Mitchell przedstawili nowe podejście do problemu SAT. Polegało ono na wprowadzeniu elementu prawdopodobieństwa do opracowywanej formuły. Ten nowy algorytm został nazwany SLS (Stochastic Local Search) [7].

Od 1994 roku badania dotyczące problemu SAT zostały przyśpieszone z powodu dużych nacisków ze strony producentów przemysłu techniki cyfrowej. Wraz z rozwojem sprzętu komputerowego istotne stało się opracowanie algorytmu, który może sprawdzić spełnialność formuły w efektywnym czasie. Należy tutaj podkreślić rozmiar takiej formuły. Formuła opisująca niektóre zagadnienia sprzętu komputerowego może zawierać milon zmiennych, zawartych w trzydziestu milonach klauzul [7].

Spośród wielu podejść do rozwiązywania problemu SAT można wyróżnić dwa główne nurty na których opierają się obecnie projektowane solvery. Pierwszy z nich polega na systematycznym przeszukiwaniu całej przestrzeni wartościowań oraz powracaniu w przypadku problemów z znalezieniem wartościowania spełniającego. Na tej podstawie opierają się algorytmy DP i DLL. Drugi z nich polega na lokalnym, stochastycznym przeszukiwaniu wartościowań. Jest to tak zwany algorytm SLS. Poniżej przedstawiam oba podejścia.

4.3 Systematyczne przeszukiwanie z powracaniem

4.3.1 Algorytm Davisa-Putnama (DP)

Ten algorytm został oparty na trzech prawach z zakresu logiki:

a) jeśli A i B są dowolnymi formułami, to koniunkcja A·(~A+B) jest równoważna z (A·B). Podobnie A·(A+B) jest równoważna z A. Dzięki temu zastosowanie reguły rezolucji (resolution) i odejmowania (subsumption) zmienia zestaw klauzul w odpowiadający mu inny, prostszy komplet.

b) jeśli X jest dowolnym zestawem formuł, a A jest badaną formuła, to X posiada model wtedy i tylko wtedy, gdy suma X i A ma model lub gdy suma X i ~A ma model.

c) przyjmijmy, że X jest zestawem badanych klauzul, a A - formułą atomową. Jeśli A nie występuje co najmniej raz jako literał pozytywny w jakiejkolwiek z klauzul zawartych w X i jednocześnie A nie występuje co najmniej raz jako literał negatywny w dowolnej z klauzul X, to usunięcie klauzul, gdzie A występuje spowoduje, że pozostały zestaw będzie posiadał model, jeśli tylko istnieje model, który spełnia X. Oznacza to możliwość usuwania tych formuł, które nie mają wpływu na wynik [14].

Poniżej została zamieszczona ilustracja, która przedstawia prostą implementację dwóch pierwszych warunków (Algorytm 1). Jako dane wejściowe funkcja **Satisfiable()** otrzymuje zestaw klauzul (clause set) S. Następnie najpierw dokonywana jest operacja odejmowania, która polega na usunięciu wszystkich klauzul zawierających L z formuły S (funkcja **deleteEveryClause(S,L)**). Po zakończeniu dochodzi do operacji rezolucji polegającej na usunięciu ~L z każdej klauzuli w formule S (**delLitFromClause(S,~L)**). Działania te trwają dopóty, dopóki dają jakikolwiek rezultat. W przeciwnym wypadku zostaje wybrany następny literał występujący w S (splitting i **chooseNextLiteral(S)**) i wywołana rekurencyjnie funkcja **Satisfiable()** kolejno dla sumy S∪L, a potem dla S∪~L. Wywołana funkcja **Satisfiable()** zatrzyma się w momencie, gdy formuła S zostanie doprowadzona do stanu, w której nie będzie zawierała żadnego literału. Wtedy S będzie spełniona. W przeciwnym wypadku S będzie niespełnione.

```
Input:        logical formula S      - formuła logiczna
Output:       SAT - satisfiable      - spełnialna

while(true){
        if(Satisfiable(S)){return(satisfiable);}
        else{return(notSatisfiable);}
}
bool Satisfiable(clause set S){
        do{
                while(unit clause L left){    // make operations for every unit clause L
                /* unit subsumption */
                S=deleteEveryClause(S,L);    // delete from S every clause containing L
                /* unit propagation */
                S =delLitFromClause(S,~L);// delete (~L) from every clause S in
                                          // which it occurs
                }
                if(S==NULL) return(satisfiable);    // when S is empty
                if(S!=NULL) return(notSatisfiable);// when clause become null in S
        }while(no changes)                    // there is no changes to do
        /* splitting */
        L=chooseNextLiteral(S)        // choose L occuring in S
                S:=S + L;             // S∪L
                if(Satisfiable(S)){return(satisfiable);}
                else{return(notSatisfiable);}
                S:=S + ~L;            // S∪~L
                if(Satisfiable(S)){return(satisfiable);}
                else{return(notSatisfiable);}
}
```

Algorytm 1 Algorytm Daviesa-Putnama

Podstawowym problemem jest w tej metodzie wybór następnego literału (część splitting). Każdy z literałów występujący w badanej jest liczony i jako pierwszy literał jest wybierany ten o najniższej wartości to znaczy najrzadziej występujący. Zamiast tak prostego sposobu wyboru można tutaj zaproponować znacznie bardziej złożone [14].

Jak można zauważyć algorytm DP nie potrzebuje specjalnego formatu wejściowego formuły. Nie trzeba sprowadzać jej do żadnej postaci normalnej. Jest to oczywista zaleta tej metody. Dzięki temu może być implementowana w „proof checkerach" czyli aplikacjach weryfikujących dowód logiczny.

Przedstawiona powyżej metoda kryje pewien potencjalny problem. Otóż przy dużych formułach może zabraknąć zasobów pamięci do kolejnych rekurencyjnych wywołań funkcji **Satisfiable()** [7]. Ten problem może zostać rozwiązany poprzez odpowiednie ograniczenia.

4.3.2 Algorytm Davisa-Logemanna-Lovelanda (DLL)

W przeciwieństwie do algorytmu DP, ten algorytm został już tak skonstruowany, aby wykorzystać zalety postaci CNF. Należy przypomnieć, że postać ta może opisywać dowolną funkcję boolowską i składa się z koniunkcji klauzul. Klauzule te zawierają alternatywy literałów czyli możliwe zmienne lub negacje możliwych zmiennych. Poważną zaletą sprowadzenia funkcji logicznej do takiej postaci jest to, że aby całe zdanie było spełnione, to każda klauzula musi być spełniona. (Na podstawie tego powstał format komputerowego zapisu formuł logicznych i jest oznaczany przez rozszerzenie *.cnf*)

Poniżej została zamieszczona ilustracja przedstawiająca omawiany algorytm (Algorytm 2). Operacja **decide()** ma za zadanie wybranie zmiennej, która nie posiada wartości i określenie tej wartości. Przypisanie wartości zmiennej nosi nazwę decyzji (decision). Przy każdym podjęciu decyzji, zostaje dodany zapis do stosu decyzji (decision stack). Jeśli funkcja nie znajdzie żadnych wolnych zmiennych, zwróci wartość false, w przeciwnym wypadku true. Brak wolnych zmiennych oznacza znalezienie wartościowania spełniającego [8].

Funkcja **bcp()** (boolean constraint propagation) identyfikuje wartościowania danej zmiennej, dzięki któremu całe badane zdanie zostanie spełnione. Działanie tej funkcji jest oparte na regule unit clause rule. Przykładowo, jeśli w pewnej klauzuli są zawarte literały o wartości 0 i jeden literał bez wartościowania, to aby klauzula była spełniona, ten wolny literał musi otrzymać wartość 1. Działanie polegające na nadaniu wartości w celu spełnienia całej klauzuli jest nazywane implikacją (implication). Tak więc działanie tej funkcji sprowadza się do sprawdzenia danej klauzuli i utworzenia odpowiednich implikacji. Jeśli nie ma już żadnych implikacji, **bcp()** zwróci wartość 1. Natomiast wartość 0 jest zwracana w przypadku konfliktu (conflict). Konflikt pojawia się wówczas, gdy dla tej samej zmiennej zostaną

utworzone implikacje, które nakazują nadanie jednocześnie wartości 0 i 1.

Należy tutaj zauważyć, że w momencie podejmowania decyzji, niektóre zmienne posiadają już wartości i są zapamiętane na stosie decyzji. Każda nowa implikacja, która zmienia wartość literału (dokonanie decyzji) jest połączona z tą decyzją i zapisywana na stosie. Dzięki temu można tę zależność zapisać jako poziom decyzji DL (decision level). Wartość DL można interpretować jako wysokość stosu w momencie generowania implikacji.

W całym algorytmie jest ważna funkcja **resolveConflict()**. Dzięki niej dokonywana jest operacja cofania się algorytmu (backtracking). Otóż w momencie powstania konfliktu dla danego literału, funkcja usuwa działanie wszystkich implikacji od implikacji wywołującej konflikt aż do pierwszej implikacji dla tego literału. Następnie jest dokonywana zmiana wartości literału na przeciwną i znowu jest uruchamiana funkcja **bcp()**. Jeśli dany literał był wypróbowany w obu stanach, funkcja szuka cofając się następnego literału nie wypróbowanego w obu stanach. Następnie znowu jest wykonywana funkcja **bcp()**. Jeśli wszystki literały zostały sprawdzone, ale nie znaleziono wartościowania spełniającego, to formuła jest niespełnialna (UN-SAT).

W przypadku tego algorytmu jest ważny wybór następnego badanego literału. Operacja ta wykonywana jest w funkcji **decide()**. Istnieje kilka róznych strategii, ktore można zaproponować. Sposoby te opierają się na testach empirycznych [8].

Istnieją zalety i wady tego algorytmu. Zaletą jest rozwiązanie problemu wykładniczego zapotrzebowania pamięci, który był wadą algorytmu DP. Natomiast wadami jest nadal wykładniczy czas rozwiązania oraz ograniczone praktyczne zastosowanie z wyłączeniem automatic proving theorem. Ponadto istnieje ograniczenie dotyczące wielkości danego problemu tj. można wykorzystać tę metodę do 1300 klauzul [7].

Przedstawiony powyżej algorytm DLL w trakcie prowadzonych badań został rozszerzony do postaci DLL-with-learning. Learning jest techniką polegającą na stworzeniu dodatkowej bazy danych, która zawiera informacje o już powziętych rozwiązaniach w trakcie przeszukiwania przestrzeni wartościowań. Poniżej zamieszczam opis tej metody (Algorytm 3).

```
Input:      logical formula       - formuła logiczna
Output:     SAT - satisfiable     - spełnialna

while(true){
        if(!decide()) return(satisfiable);          //if no unassigned vars
        while(!bcp()){
        if(!resolveConflict()) return(notSatisfiable);
        }
}

bool resolveConflict(){
d=most recent decision not „tried both ways"
if(d==NULL) return(false); //no such d was found

/* operation with d */
flip the value of d;
mark d as tried both ways;
undo any invalidated implications;
return(true);
}
```

Algorytm 2 Algorytm Daviesa-Longemana-Lovelanda (DLL)

Funkcja **decide_next_branch()** ma za zadanie wybrać następną zmienną. Wybór jest dokonywany na podstawie różych heurystyk. Każda taka decyzja ma przypisaną zmienną, która opisuje poziom decyzji. Funkcja **deduce()** analizuje zmianę dokonaną przez wybór danego literału i wykrywa ewentualne konflikty *(conflict)*. Konflikt oznacza sytuację, w której wybrana zmienna otrzymuje sprzeczną wartość z wcześniej ustaloną. Wtedy wywoływana jest funkcja **analyze_conflicts()** w celu analizy i rozwiązania powstałego przypadku. W tym celu następują badania tej klauzuli oraz innych, które są powiązane przez wspólne zmienne. Jako efekt tej pracy funkcja zwraca wartość opisującą do jakiego poziomu decyzji należy się cofnąć, aby rozwiązać konflikt. Jeśli funkcja zwróci 0, to oznacza, że formuła jest niespełnialna. Poziom decyzji, efekt analizy, jest argumentem dla funkcji **backtrack()**. Zadaniem jej jest cofnięcie błędnych decyzji w celu rozwiązania konfliktu [15].

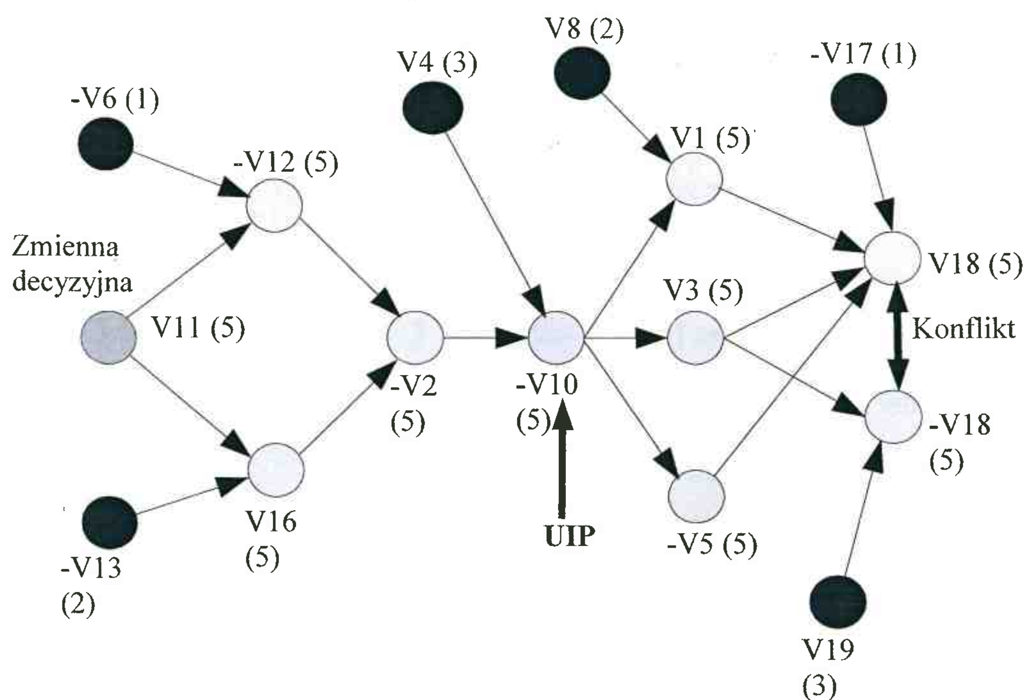
```
Input:      logical formula        - formuła logiczna
Output:     SAT - satisfiable      - spełnialna

while(true){
        if(decide_next_branch()){                              // Branching
                while(deduce()==conflict){                     // Deducing
                        blevel = analyze_conflicts();          // Learning
                        if(blevel==0){ return (notSatisfiable);}
                        else {back_track(blevel); }            // Backtracking
                }
        else {return(satisfiable);}   //no branch = all variables got assigned
        }
}
```

Algorytm 3 Algorytm DLL-with-learning

W porównaniu z metodą DLL przedstawiany algorytm jest wzbogacony o rozbudowany backtracking. Polega on na tym, że algorytm może cofnąć się o kilka poziomów decyzji (non-chronological backtracking). Ponadto w trakcie wykonywania tej operacji, można zachować informacje o przyczynach sytuacji konfliktu (learning). W ten sposób algorytm nie będzie sprawdzał podobnych sytuacji w dalszym przeszukiwaniu przestrzeni wartościowań.

W celu zobrazowania zależności pomiędzy zmiennymi, ich wartościowaniem a wpływem na tworzenie się konfliktu w trakcie procesu rozwiązywania problemu jest tworzony graf implikacji (implication graph). Jest to graf skierowany i acykliczny (DAG - directed acyclic graph). Każdy wierzchołek reprezentuje zmienną i jej wartość. Natomiast krawędzie incydentne obrazują powody danego wartościowania. Wierzchołek decyzji (decision vertex) nie posiada krawędzi incydentnej. Ponadto do każdej zmiennej jest przypisany poziom decyzji (w postaci liczby w nawiasach). Jeśli nie ma konfliktu, to dla każdej zmiennej występuje jeden wierzchołek. Konflikt jest zobrazowany poprzez istnienie dwóch wierzchołków dla jednej zmiennej. Powyżej przedstawiam przykładowy graf implikacji (Ilustracja 1) [15].



Ilustracja 1 Graf implikacji

W grafie implikacji wierzchołek 'a' dominuje nad wierzchołkiem 'b' wtedy i tylko wtedy, gdy każda droga z zmiennej decyzyjnej na tym samym poziomie decyzyjnym co 'a', aby dojść do 'b' potrzebuje przejść przez 'a'. A więc Unikalny Punkt Implikacji (UIP – Unique Implication Point) to wierzchołek, który dominuje na danym poziomie decyzji nad wierchołkami zmiennej, która spowodowała konflikt. Przykładowo na Ilustracji 1 V10, V2 i V11 są UIP. Można określić, że UIP jest pojedynczą przyczyną konfliktu na danym poziomie decyzji, dlatego też zasadne jest zaczynać analizę od UIP. Należy tu zauważyć, że UIP jest zawsze zmienną decyzji (decision variable) oraz może być kilka UIP dla danego konfliktu (patrz Ilustracja 1).

W efekcie tak rozwinięta metoda obrazowania konfliktu po zaimplementowaniu algorytmu umożliwia dwie rzeczy. Pierwsza z nich to możliwość cofania się o więcej niż jeden poziom decyzji (non-chronological backtracking).Informacja na który poziom decyzji DL należy się cofnąć wynika z analizy i jest to najwcześniejszy poziom. Druga z nich - możliwość uczenia (learning), ponieważ w trakcie badania konfliktu oraz wnioskowania poziomu decyzji na który trzeba się przejść, można stworzyć specjalną bazę danych, która pozwala przechować informację o przyczynach danego konfliktu i pozwolić na ominięcie tego błedu w póżniejszej fazie. W tym przypadku

należy utworzyć klauzulę konfliktu (conflict clause), która będzie zapisem informacji o wystąpionym konflikcie i dodać ją do specjalnej bazy danych. Tworzenie takiej klauzuli jest wynikiem analizy i podziału grafu implikacji.

W trakcie trwania badań powstało kilka różnych schematów nauki (learning scheme). Przykładowo, w momencie utworzenia sytuacji konfliktu, jest budowany graf implikacji. Następnie jako klauzulę konfliktu zapisywana jest klauzula składająca się z wszystkich zmiennych występujących w grafie prócz zmiennych decyzji. Taki schemat jest nazywany schematem „Decision".

Z całą pewnością da się stwierdzić, że dobry schemat powinien zredukować liczbę decyzji potrzebnych do rozwiązania problemu. Istnieje przekonanie, że im krótsza jest klauzula, która jest wnioskiem analizy konfliktu, tym lepszy schemat nauki. Jednakże należy zauważyć, że o użyteczności i efektywności danego schematu nauki można jedynie powiedzieć jedynie na podstawie danych empirycznych. Badania teoretyczne nie są miarodajne.

4.3.4 Heurystyki

Pomimo istnienia kilku algorytmów, o których mowa powyżej, są one podobne, ponieważ opierają się na tej samej zasadzie przeszukiwania całej przestrzeni wartościowań w celu stwierdzenia spełnialności danej formuły. We wszystkich wyżej przedstawionych metodach jest ważne, aby efektywnie wyszukiwać kolejne zmienne oraz wyliczać ich wartościowania. Jest to istotne, ponieważ w tym punkcie algorytmu (przykładowo **decide()** w DLL) można przyśpieszyć rozwiązanie zadania poprzez wybieranie tych zmiennych, które na podstawie dokonanych wcześniej badań, są bardziej obiecujące tj. ich badanie pozwala na szybsze przedstawienie rozwiązania.

Tak jak to już zostało wcześniej zaznaczone, brak efektywnego algorytmu dla problemu SAT i jednocześnie zapotrzebowanie na rozwiązania przykładów problemu SAT pochodzących z różnych dziedzin nauki i przemysłu zaowocowało wprowadzeniem rozwiązań opartych na heurystykach. Heurystyki można przedstawić jako podejście intuicyjne w teorii algorytmów. Większość heurystyk powstaje w wyniku analizy danych empirycznych. Również przewagi jednej nad drugą są oparte na badaniach i prowadzonych testach. Często także nie istnieją dowody matematyczne na

skuteczność danej heurystyki. W przypadku solverów SAT heurystyki są używane do wyboru następnej zmiennej oraz jego wartościowania.

W trakcie prac nad solverami SAT zostały wypracowane różne heurystyki. Poniżej przedstawiam krótki przegląd najpopularniejszych :

a) RAND - polega na przypadkowym wybraniu zmiennej, która nie ma jeszcze żadnej wartości, a następnie na przypisaniu jej wartości również wybranej przypadkowo [11].
b) MOMS - polega na wybraniu zmiennej, która występuje w największej liczbie klauzul o najmniejszej długości (niezależnie od negacji lub braku). Następnie należy wybrać takie wartościowanie, które spełnia więcej klauzul [11].
c) BOHM - polega na wybieraniu tych literałów, które albo spełniają więcej klauzule (należy nadać wartość 1) albo przy nadaniu wartości 0 nie powodują zmiany wartości całej klauzuli prócz liczby literałów bez wartości [11].
d) Jeroslow-Wang - wybierane są te zmienne, które występują w wielu małych klauzulach i mogą przyjąć jedną wartość (tzn. JW-OS) lub wybierane są te zmienne, które mogą przyjąć dowolną wartość również występujące w wielu klauzulach (tzn. JW-BS). Wartość wybranej zmiennej jest dobierana tak, aby spełniła największą ilość klauzul [11].
e) DLS - jest oparte na wyborze zmiennej, która dla tej samej wartości występuje w największej liczbie klauzul (DLIS) lub występuje w największej liczbie klauzul niezależnie od wartości (DLCS). Wartość logiczna wybranej zmiennej powoduje spełnienie się maksymalnej liczby klauzul. Istnieje wersja przypadkowego wyboru wartości dla wybranej zmiennej (RDLCS) [11].
f) VSIDS - polega na utworzeniu specjalnego licznika zliczającego każdy literał w zależności od polaryzacji. Licznik jest inkrementowany w momencie dodawania nowej klauzuli. Jako następna zmienna jest wybierana ta, która ma największą wartość licznika i nie ma przypisanej wartości. W przypadku dwóch zmiennych o tej samej wartości licznika, wybór jest dokonywany przypadkowo. Ponadto okresowo wartość licznika jest dzielona przez pewną określoną stałą [8].
g) BerkMin - polega na utworzeniu listy ostatnio analizowanych klauzul, które nie zostały spełnione. Podobnie utworzona jest lista zmiennych, które najczęściej występowały w trakcie analizy konfliktu, wraz z licznikiem zliczającym częstość występowania. Jako następna zmienna wybierana jest ta o najwyższym numerze.

W przypadku braku niespełnionych klauzul i w rezultacie wykorzystaniu wszystkich zmiennych wybór jest dokonywany na podstawie dodatkowej heurystyki globalnej [7].

4.4 Stochastyczne przeszukiwanie lokalne

4.4.1 Algorytm SLS

Rozwój algorytmów opartych na SLS (*stochastic local search*) przypada na lata 90-te ubiegłego wieku. Pierwsze publikacje przypadają na lata 1990-1992. Pod koniec ubiegłego wieku i na początku tego osiągnięto znaczący rozwój tej metody rozwiązania problemu SAT. Ważną cechą tego algorytmu jest to, że niemożliwe jest określenie niespełnialności formuły. Po wykonaniu wszystkich działań tego algorytmu mogą zostać podane dwa wyniki dla podanej formuły : albo formuła jest spełniona albo spełnialność nie została dowiedziona. Druga informacja nie przesądza o tym, że podana formuła jest niespełnialna. Oznacza jedynie tyle, że nie zostało znalezione wartościowanie spełniające [13].

Poniżej przedstawiam opis tej metody (Algorytm 4). Jako argumenty tej funkcji podawane są trzy parametry: formuła w formacie CNF, liczba *maxTries* oraz liczba *maxSteps*. W rezultacie można otrzymać wartościowanie dla podanej formuły albo informacje o nie znalezieniu rozwiązania.

```
Input:      logical formula CNF-formula        - formuła logiczna
Output:     SAT - satisfiable                  - spełnialna

bool LocalSearch(CNF-formula f, maxTries, maxSteps){
        for(i=1; i<maxTries; i++){
        s=initAssign(f);
                for(j=1; j<maxSteps; j++){
                        if(satisfies(s,f)){return(satisfiable);}
                        else{
                        x = chooseVariable(f,s);
                        s = s with truth value flipped for x;
                        }
                }
        }
        return(notSatisfiable);         // no solution found
}
```

Algorytm 4 Algorytm Stochastic Local Search (SLS)

Przedstawiony algorytm zaczyna pracę z pewnym losowo wybranym

wartościowaniem (funkcja **initAssign()**). Następnie jest sprawdzane, czy wartościowanie rozwiązuje tą formułę (**satisfies()**). Jeśli nie, wykonywana jest operacja mająca na celu doprowadzenie wybranego wartościowania do takiego stanu, które rozwiąże daną formułę. Dzieje się to poprzez następującą iterację: wybór zmiennej (funkcja **chooseVariable()**), zmiana wartości tej zmiennej na przeciwną, sprawdzenie spełnialności nowego wartościowania. Te czynności mogą być powtarzane w nieskończoność i nie dać efektu. Dlatego też liczba iteracji jest ograniczona do *maxSteps*. Po tym jest losowane kolejne wartościowanie i znowu dochodzi do sprawdzenia spełnialności. Wybieranie nowych, losowych wartościowań również jest ograniczone do liczby *maxTries*.

4.4.2 Heurystyki

W przypadku tego algorytmu bardzo dużą rolę odgrywa podejście heurystyczne, które występuje w funkcji **chooseVariable()**. Tylko w tym miejscu można przyspieszyć działanie algorytmu poprzez wybieranie efektywnych zmiennych. Istnieje kilka heurystyk z powodu swej wartości w algorytmie traktowanych jak strategie:

a) GSAT - polega na tym, że w każdym kroku zmieniana jest ten literał, który powoduje maksymalne zwiększenie (lub minimalne zmniejszenie) spełnionych klauzul. W przypadku dwóch lub więcej zmiennych, które mogą zostać wybrane, wybór jest dokonywany losowo.
b) Random Walk - polega na losowym wyborze zmiennej i zmianie jej wartości występującej w obecnie badanej niespełnionej klauzuli.
c) GWSAT - polega na wybieraniu pomiędzy GSAT a Random Walk w kolejnych krokach z pewnym prawdopodobieństwem *wp* (*noise*).
d) SKC - polega na wyborze tych literałów, które spowodują minimalną zmianę liczby spełnionych klauzul na niespełnione. Jeśli jest możliwe zachowanie stałej liczby spełnionych klauzul, należy wybrać te literały. W przeciwnym przypadku zmienna jest wybierana z pewnym prawdopodobieństwem *wp*.

e) Tabu - należy wybrać literał, którego zmiana spowoduje maksymalne zwiększenie się liczby spełnionych klauzul. Ponadto należy utrzymywać listę o stałej długości ostatnio zmienionych literałów. W przypadku dwóch wybranych zmiennych, wybór należy dokonać losowo.

f) Novelty - polega na ułożeniu listy literałów, których zmiana spowoduje maksymalny przyrost spełnionych klauzul. Jeśli najlepsza zmienna na tej liście, nie została jeszcze zmieniona, zawsze należy wybrać tą drogę. W przeciwnym wypadku, należy wybrać drugi-najlepszy wybór na liście z prawdopodobieństwem *wp*.

g) R-Novelty - jest podobny do poprzedniej heurystyki. Najpierw dokonywany jest tworzona lista zmiennych w klauzulach w zależności od tego, jak zmiana danej zmiennej wpłynie na liczbę spełnionych klauzul tj. czy zwiększy czy zmniejszy tą liczbę. W przypadku dwóch zmiennych, które są najlepsze, należy wybrać tą, która pochodzi z rzadziej zmienianej klauzuli. W przeciwnym wypadku istnieje najlepszy wybór i drugi-najlepszy wybór. Jeśli najlepszy wybór pochodzi z rzadziej zmienianej klauzuli, należy wybrać tą zmienną. Drugi-najlepszy literał jest wybierany, gdy istnieję powody, aby nie wybierać pierwszej zmiennej [6].

h) WalkSAT - polega na losowym wyborze jeszcze niespełnionej klauzuli, następnie wybór literału jest dokonywany zgodnie z jedną uznanych heurystyk.

i) R-Novelty$^+$ i Novelty$^+$ - te heurystyki zostały rozszerzone o Random Walk. W trakcie badań okazało się, że R-Novelty i Novelty w pewnych określonych wypadkach mogą wejść w pracę cykliczną i zatrzymać się. Dzieje się tak w momencie, kiedy dochodzi do sprawdzania i zmiany w klauzulach, które już zostały sprawdzone. Dlatego też wprowadzono heurystykę Random Walk, która jest wybierana co n kroków [6].

5 Analiza działania wybranych solverów SAT

5.1 Wybór solverów SAT

Badanie i prace nad solverami SAT jest domeną uniwersyteckich instytutów naukowych. Istnieją oczywiście rozwiązania komercyjne, które są wykorzystywane w wielu dziedzinach przemysłu. Jednak większość solverów jest wynikiem prac naukowców. Dzięki temu istnieje możliwość korzystania z nich, ponieważ ich kody źródłowe są publikowane.

Dla celów tej pracy wybrałem trzy solvery SAT. Są to zChaff, Unitwalk i march_eq. Poniżej przedstawiam powody takiego wyboru.

a) zChaff - jest to aplikacja napisana przez zespół naukowców z uniwersytetu Princeton, szczególnie Yinlei Yu i Lintao Zhang. Opiera się ona na aplikacji Chaff, który powstał w wyniku połączonych prac naukowców, takich jak M.W. Moskewicz i S. Malik, z Princeton, MIT i Berkeley. ZChaff jest napisany w języku C++ dla systemu operacyjnego Unix/Linux. Analizowana wersja jest wersją z końca 2004 roku, więc zawiera najnowsze rozwiązania i możliwości. Jednak głównym powodem wyboru tej aplikacji było to, że jest oparta na algorytmie *DLL-with-learning*. Jednocześnie ta aplikacja jest podstawą wielu innych solverów. Niektóre zespoły naukowców korzystają z opublikowanych rozwiązań, aby przykładowo zająć się pewnymi szczegółowymi problemami. Takimi solverami są min. CompSat oraz limmat. W 2002 r. zChaff zdobył pierwszą nagrodę *„SAT Contest '02"* w kategorii formuł pochodzących z zastosowań przemysłowych.

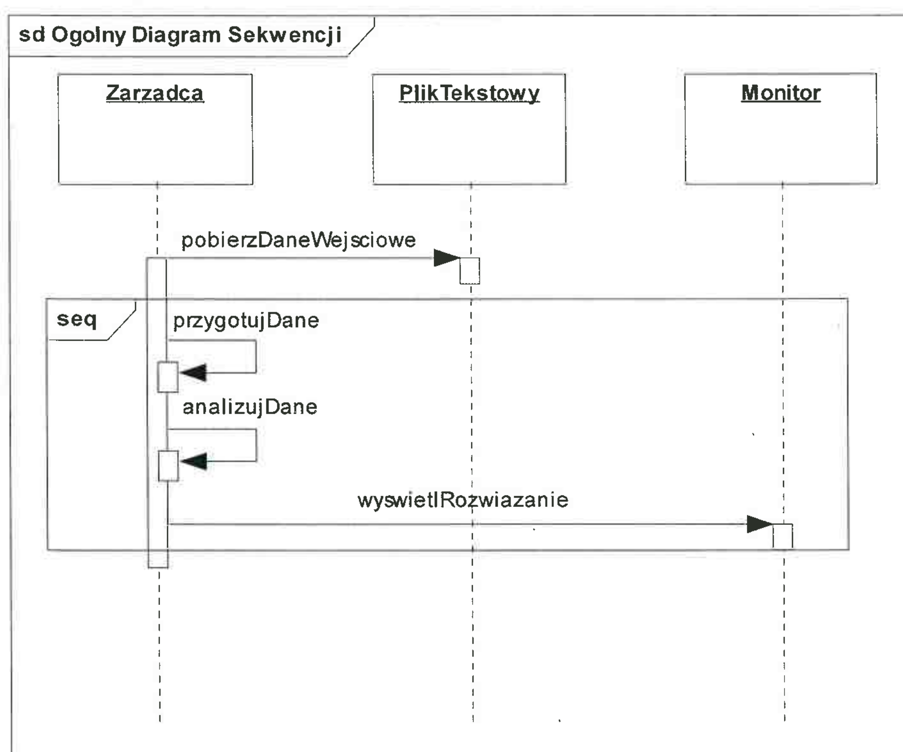
b) UnitWalk - jest aplikacją napisaną przez E.A. Hirscha i A. Kojevnikova. Jest to przykład zastosowania algorytmu SLS. Na jego podstawie postaram się omówić ten drugi typ możliwych aplikacji. Aplikacja została napisana w języku C dla platformy Unix/Linux. Po raz pierwszy została zaprezentowana w 2002 roku. Należy tu zauważyć, że nie jest to jedyna aplikacja tego typu. Istnieje wiele innych przykładowo GSAT. Niemniej ta jest jedną z bardziej rozwiniętych i dostępnych.

c) march_eq - trzeci, badany solver jest jedną z najnowszych aplikacji pod względem rozwiązań i optymalizacji. Został napisany w 2001 r. przez M. Dufoura, M.

Heule'a i J. van Zwietena. Program wykonano w języku C dla systemu operacyjnego typu Unix/Linux. Jest to przykład implementacji algorytmu DLL, ale o znacznie lepszych parametrach min. o przyśpieszonym działaniu i lepszym zaprojektowaniu kodu. Dowodem tego jest zdobycie pierwszej nagrody w kategorii formuł przygotowanych przez organizatora w trakcie ostatnich *„SAT Contest '04"*. W innych kategoriach ten solver również zajął wysokie pozycje.

5.2 Analiza działania solvera zChaff

Pierwszym, badanym solverem jest zChaff. Ogólny zarys działania tej aplikacji przedstawia diagram sekwencji (Ilustracja 2)



Ilustracja 2 Ogólny diagram sekwencji solvera zChaff

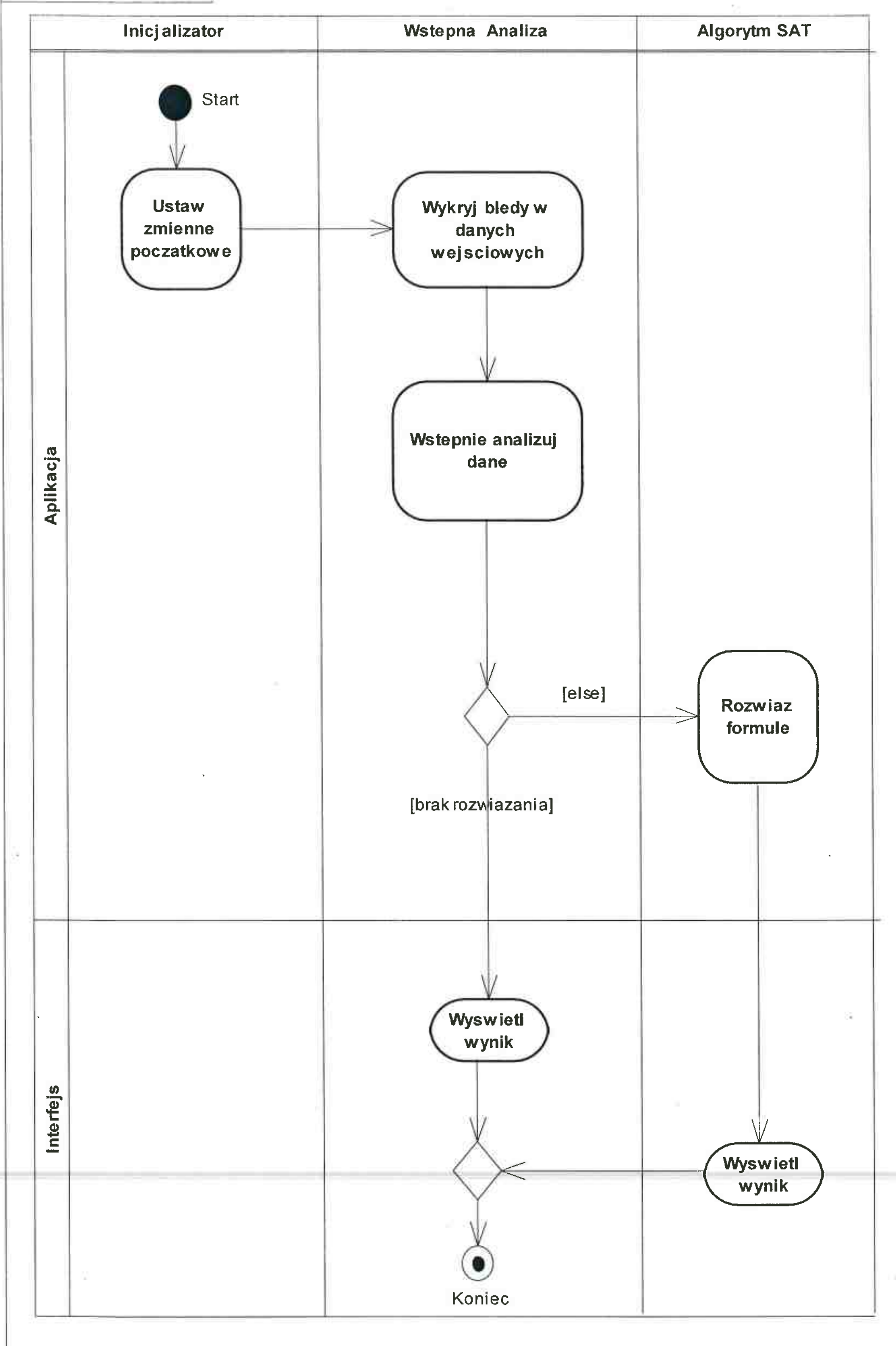
Ogólny diagram sekwencji przedstawia zależności pomiędzy trzema elementami, które współdziałają w tej aplikacji. **Zarządca** jest to główna część solvera zChaff. Jego zadaniem jest pobranie danych wejściowych czyli formuły z pliku tekstowego (**PlikTekstowy**). Następnie dokonanie niezbędnych operacji na formule w celu uzyskania wartościowania spełniającego. Natomiast **Monitor** służy do wyświetlenia wyników działań Zarządcy.

Fragment z słowem kluczowym **seq** wskazuje na to, że te operacje są słabo uszczegółowione. Dlatego też ta część aplikacji została rozszerzona w diagramie aktywności (Ilustracja 3).

Diagram aktywności został podzielony według dwóch kryteriów. Pierwszym z nich jest podział na części zgodnie z odpowiedzialnością wyróżnionych elementów czyli **Aplikacja** i **Interfejs**. **Aplikacja** jest odpowiedzialna za przetwarzanie danych wejściowych i uzyskanie wyników. Wyniki są przedstawiane w części **Interfejsu**. Drugie z kryteriów to podział zgodnie z rolami, które działają. Zostały wyróżnione trzy role. Zadaniem **Inicjalizatora** jest przygotowanie środowiska aplikacji do działań związanych z rozwiązywaniem formuły. Jest to związane z inicjalizacją pamięci i ustawieniem wartości zmiennych. **Wstępna Analiza** pozwala na sprawdzenie poprawności badanej formuły i jej uproszczenie. W trakcie tych operacji jest możliwe wykazanie niespełnialności formuły czyli braku rozwiązania. Następnie po pomyślnym zakończeniu wstępnej analizy, jest wykonywana czynność rozwiązywania formuły w części **Algorytm SAT**. Po zakończeniu tych operacji wynik jest przedstawiany na monitorze.

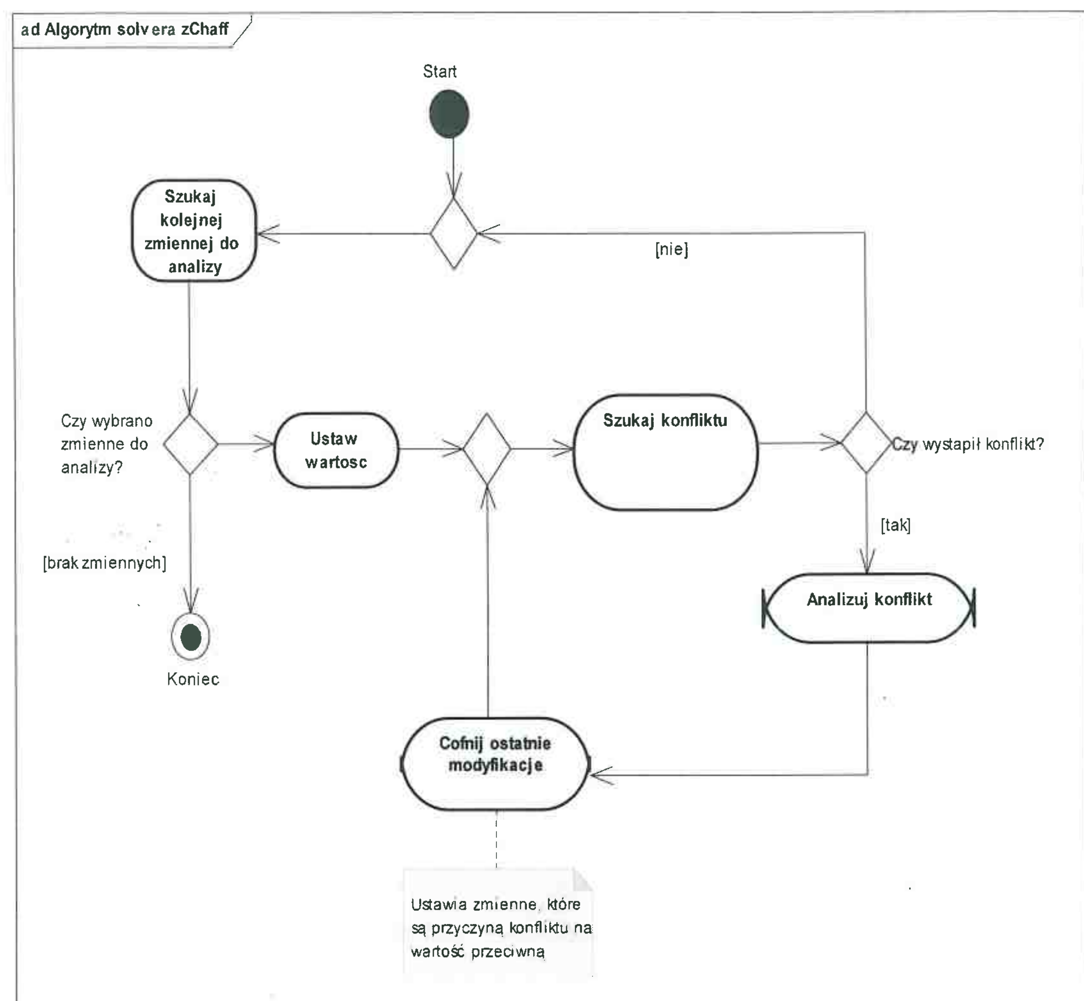
Najważniejszą częścią solvera jest część w której jest rozwiązywana formuła. Ta część ze względu na swoją złożoność została przedstawiona w postaci diagramu aktywności (Ilustracja 4). Jednocześnie na tym diagramie został przedstawiony algorytm DLL-with-learning w języku UML 2.0.

Pierwszą czynnością tego algorytmu jest wybranie kolejnej zmiennej do analizy. W przypadku braku takich zmiennych dochodzi do zakończenia pracy, ponieważ oznacza to wyznaczenie wartościowania dla wszystkich zmiennych. Jeżeli zostanie wybrana zmienna, to następnym krokiem jest ustawienie jej wartości. Należy przy tym sprawdzić ewentualne wystąpienie konfliktu. W przypadku braku konfliktu, zostaje wybrana następna zmienna i są wykonywane kolejne czynności. W przypadku



Ilustracja 3 Szczegóły solvera zChaff

wystąpienia konfliktu, jest wykonywana analiza przyczyn, w tym momencie są zbierane informacje na podstawie, których dochodzi do operacji *learning*. Potem jest wykonywane cofnięcie wszystkich ostatnich modyfikacji, dzięki czemu jest możliwe rozwiązanie konfliktu. Po zakończeniu tych czynności ponownie jest sprawdzane wystąpienie konfliktu. Dzięki temu jest uzyskiwana poprawność wymienionych wyżej zadań.



Ilustracja 4 Algorytm solvera zChaff

Na ilustracji 4 można zauważyć nowe elementy diagramu aktywności. Pierwszym z nich jest zupełnie inne wykorzystanie węzłów. Zgodnie z nowym standardem UML 2.0 węzły mogą być użyte w dwóch funkcjach:

a) węzeł połączenia (merge node) - jest to punkt, gdzie spotykają się krawędzie incydentne z poszczególnymi polami aktywności. Dzięki temu jest możliwe zobrazowanie niuansów zależności pomiędzy poszczególnymi

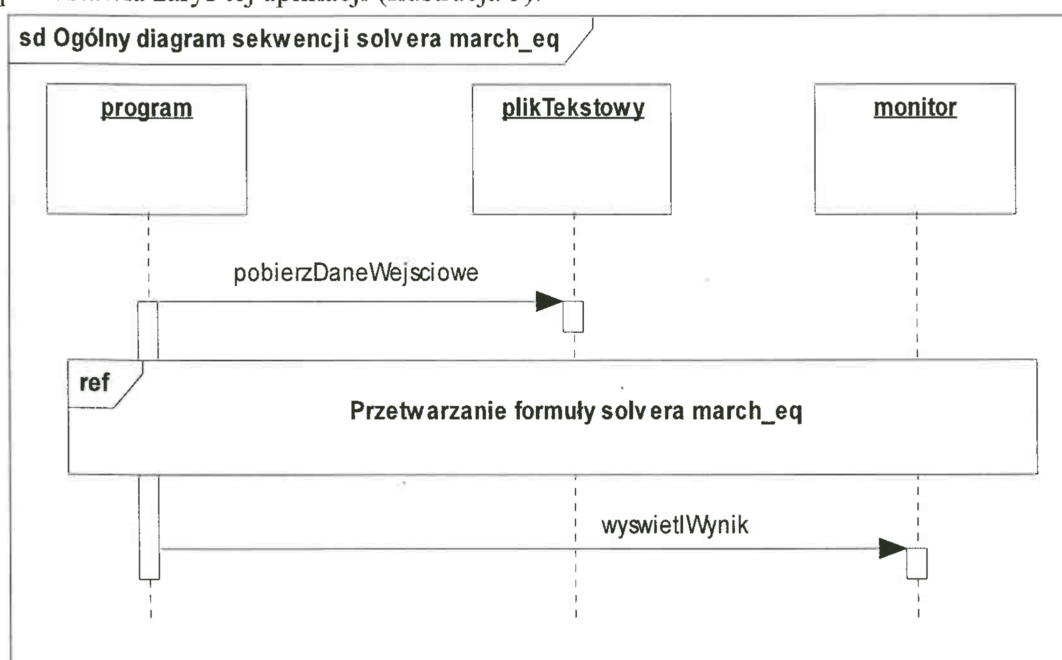
czynnościami.

b) węzeł decyzyjny (decision node) - jest to punkt, w którym dochodzi do rozgałęzienia się ścieżki zgodnie z podjętą decyzją (podobnie jak w konstrukcji if..else..then).

Drugą zmianą wprowadzoną w diagramie aktywności jest zastosowanie przy modelowaniu semantyki sieci Petrie'go. W rezultacie jest zasadne wykorzystanie węzłów w sposób przedstawiony powyżej [10].

5.3 Analiza działania solvera march_eq

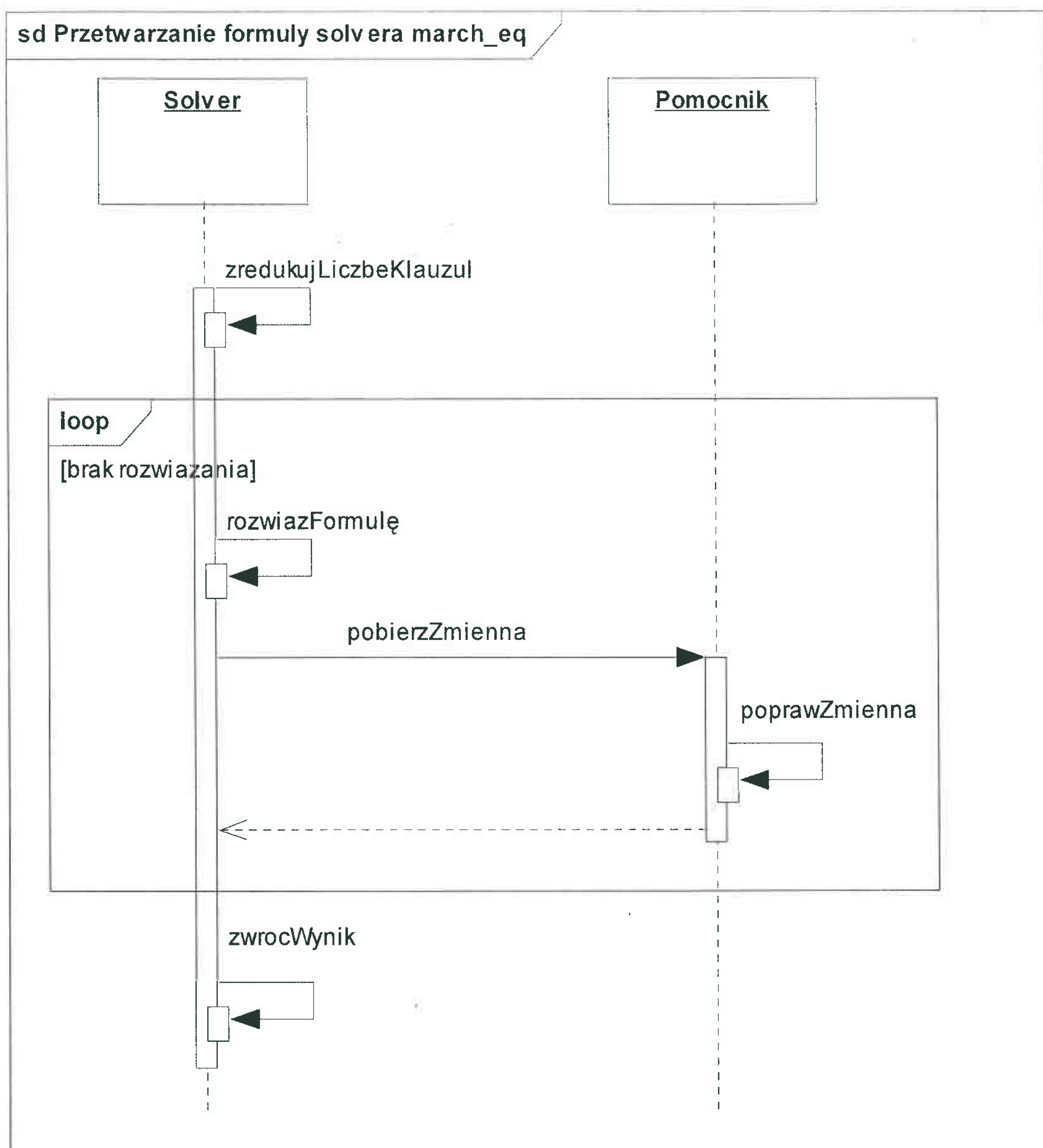
Następnym analizowanym solverem jest march_eq. Ogólny diagram sekwencji przedstawia zarys tej aplikacji (Ilustracja 5).



Ilustracja 5 Ogólny diagram sekwencji solvera march_eq.

Wyróżnione zostały trzy elementy, które współdziałają ze sobą. **Program** jest określeniem aplikacji tj. jej działań. W stosunku do niej jako zewnętrzne elementy pozostają **plikTekstowy** i **monitor**. W trakcie działania aplikacji na początku dochodzi do pobrania badanej formuły, natomiast pod koniec jest wyświetlany wynik w sposób umożliwiający użytkownikowi jego analizę. Fragment oznaczony słowem kluczowym **ref** wskazuje na wykorzystanie referencji do innego diagramu, który w tym przypadku został przedstawiony na ilustracji 6.

Ilustracja 6 Przetwarzanie formuły solvera march_eq



Na ilustracji 6 jest opisana zależność pomiędzy dwoma elementami aplikacji. **Solver** jest główną częścią aplikacji. Odpowiada za przygotowanie formuły poprzez redukcję liczby klauzul i przetworzenie do postaci 3-CNF. Do jego zadań należy również przesłanie wyniku do części, która wyświetla na monitorze. Jednak najważniejsza część tego diagramu jest wyznaczona przez fragment z słowem kluczowym loop. Słowo kluczowe loop oznacza wykonywanie pętli dopóty, dopóki jest spełniany podany warunek, w tym przypadku „brak rozwiązania". W wyróżnionej pętli jest pokazana cała specyfika tego solvera. Otóż solver ten próbuje rozwiązać formułę.

W sytuacji, gdy nie może tego dokonać z powodu określonych zmiennych jest wywoływana część **Pomocnik**, której zadaniem jest rozwiązanie problemów ze zmienną. Po pomyślnym rozwiązaniu tego problemu dochodzi do ponownego wywołania części odpowiedzialnej za rozwiązywanie formuły. Tak więc dochodzi tutaj do wykorzystania rekurencji w celu przyśpieszenia pracy solvera.

5.4 Analiza działania solvera UnitWalk

Trzecim wybranym solverem jest UnitWalk. Ogólne działanie tego solvera zostało przedstawione z zastosowaniem diagramu współdziałania (interaction diagram) (Ilustracja 7). Ten rodzaj diagramu łączy ze sobą elementy diagramu aktywności i sekwencji. Z tym, że węzłami są referencje do innych diagramów lub fragmenty diagramów tj. egzemplarze interakcji (interaction occurences) [10].

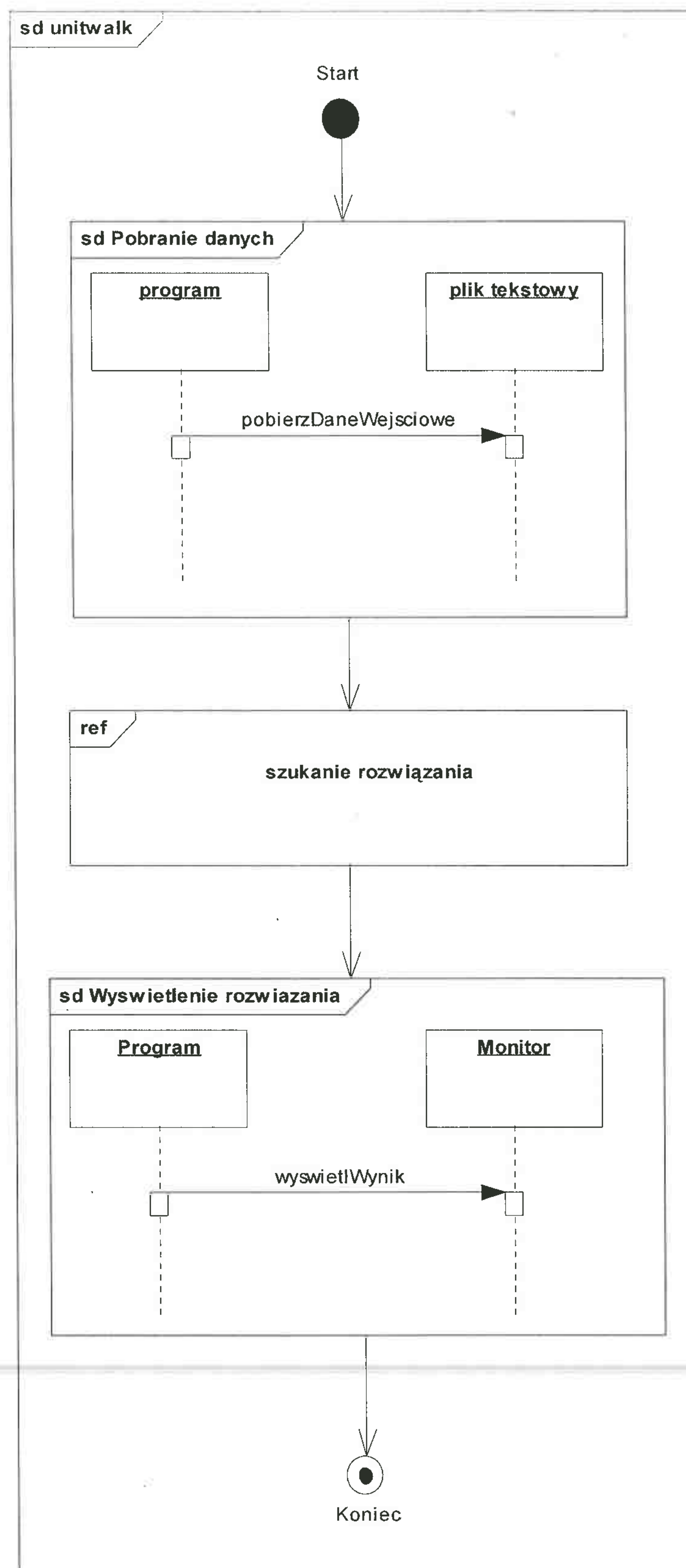
Na omawianej ilustracji pobierane danych i wyświetlanie wyniku zostało przedstawione za pomocą fragmentów innych diagramów. Tak jak poprzednio głównym celem takiego diagramu jest czytelność. Dodatkowo została wykorzystana referencja do diagramu, który przedstawia szczegóły szukania rozwiązania. Ta część solvera została przedstawiona przy pomocy diagramu aktywności (Ilustracja 8).

Pierwszą czynnością na tym diagramie jest inicjalizacja danych. Następnie rozpatrywane jest pytanie zawarte w węźle decyzyjnym. Treść pytania wynika ze specyfiki algorytmu SLS. Natomiast wykorzystanie notki, która wskazuje na dany węzeł pozwala na przedstawienie długiego pytania przy pomocy słowa kluczowego **<<decisionInput>>**. W rezultacie pytanie nie jest powtarzane na poszczególnych gałęziach, co zwiększa czytelność diagramu. Jest to jeszcze jeden nowy element wprowadzony w języku UML 2.0.

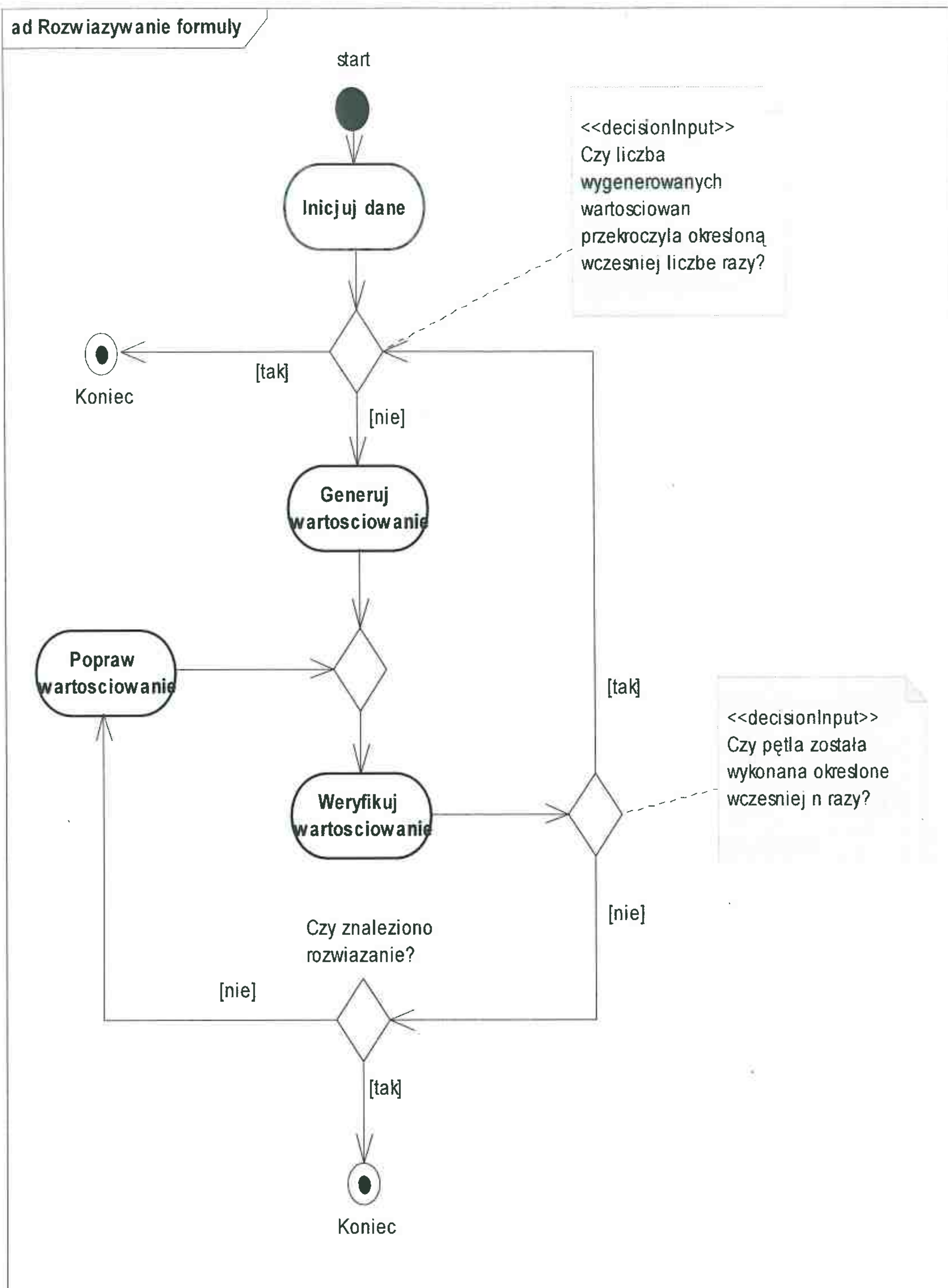
Następną czynnościa aplikacji jest wygenerowanie wartościowania dla badanej formuły. Generowanie przebiega całkowicie losowo. Utworzone wartościowanie jest weryfikowane pod kątem spełnialności z formułą. Po wykonaniu tej czynności jest badana liczba iteracji. Następnie w przypadku, gdy nie przekroczono wcześniej założonej liczby powtórzeń, wynik weryfikacji decyduje o dalszych czynnościach. Sytuacja w której nie dochodzi do spełnienia formuły przez wylosowane wartościowanie, powoduje, że dane wartościowanie w losowy, heurystyczny sposób jest poprawiane. Nowa wersja wartości zmiennych jest weryfikowana. Czynności te są

powtarzane, dopóki nie zostanie znalezione wartościowanie spełniające bądź nie wyczerpie liczba pętli. W tym drugim przypadku, zostanie wygenerowane zupełnie nowe wartościowanie, które będzie badane n razy. Liczba możliwych wygenerowanych wartościowań również jest ograniczona. W przeciwnym razie jest możliwe działanie takiej aplikacji w nieskończoność, ponieważ algorytm SLS nie ma możliwości potwierdzenia niespełnialności danej formuły.

Na przedstawionym diagramie można zauważyć dwa stany końcowe. Wskazują one to samo miejsce w aplikacji. Drugi stan końcowy został wprowadzony w celu zwiększenia czytelności całego szkicu.



Ilustracja 7 Ogólny diagram sekwencji solvera UnitWalk



Ilustracja 8 Diagram aktywności rozwiązywania formuły

5.5 Podsumowanie

Analiza działania wybranych solverów, która została przedstawiona powyżej pozwala na wyciągnięcie kilku wniosków. Niezależnie od rodzaju badanej aplikacji, wszystkie posiadały wspólne elementy. Dane wejściowe były pobierane z specjalnych plików tekstowych. Wyniki, zarówno pozytywne jak i negatywne były wyświetlane na ekranie. Występowały różnice pomiędzy sposobem przedstawiania tych informacji. Ponadto, oprócz solvera UnitWalk, pozostałe aplikacje zawierały część, której zadaniem było wstępne uporządkowanie badanej formuły i jej uproszczenie. Wszystkie solvery przed rozpoczęciem analizy inicjalizowały środowisko pracy tj. alokowały pamięć i ustawiały zmienne pracy. W efekcie z powyższych analiz wynika, że budowa badanych solverów na przedstawionym powyżej poziomie abstrakcji jest podobna. Różnią się one jednak głównym algorytmem, który rozwiązuje problem SAT.

Diagramy, które opisywały badane solvery, zostały przygotowane na podstawie opublikowanych kodów źródłowych. Należy tutaj podkreślić, że w trakcie analizy programów dokonano operacji zbudowania projektu na podstawie zależności aplikacji. Jest to tzw. reverse engineering. Zwykle najpierw jest przygotowywany projekt aplikacji, a potem zostaje wykonany program. W tej kolejności jest to nazywane forward engineering.

Diagramy zostały przygotowane zgodnie z najnowszymi specyfikacjami języka UML tj. wersji 2.0. Dzięki nowym elementom takim jak fragmentacja, semantyka PetriNet można było z powodzeniem zbudować modele funkcjonowania algorytmów, co we wcześniejszych wersjach języka było nie do zrealizowania.

6. Porównanie wybranych solverów SAT

6.1 Wybór narzędzia analitycznego

Solvery SAT są aplikacjami, które są budowane w języku C lub C++. Jest to związane z wydajnością i szybkoscią takiego kodu. Ponadto większość jest przygotowana w oparciu o system operacyjny Linux/Unix, ponieważ najczęściej powstają jako projekty akademickie na takich uczelniach jak Princeton lub MIT itp. Niektóre programy są zaprojektowane w taki sposób, aby umożliwić kompilację przez programy działające w systemie Windows, jednak należą one do mniejszości. Powyższe założenia pokazują, jakie warunki powinny spełniać narzędzia testujące. Oznacza to, że należy przyjąć ograniczenie dotyczące platformy (Linux/Unix) i języka programowania (C lub C++).

Jedną z cennych możliwości systemu operacyjnego Linux jest ogólnodostępne oprogramowanie na licencji OpenSource. W trakcie szukania narzędzi do testowania aplikacji znalazłem wiele programów sprawdzających sieci komputerowe, jakość stron HTML lub baz danych. Znalazłem też kilka projektów wspierających testowanie oprogramowania. W rezultacie z tego typu rozwiązań mogę przedstawić takie aplikacje jak : cpptest, cutee lub TUT. Mają one jedną wspólną wadę. Pozwalają na jednostronne testowanie produktu tzn. najczęściej są to programy sprawdzające dostęp i zarządzanie pamiecią. Nie są uniwersalnym środowiskiem do przygotowywania aplikacji.

W trakcie szukania odpowiedniego narzędzia testującego znalazłem propozycje firmy Rational, która zajmuje się przygotowywaniem narzędzi pozwalających na produkcję wysokiej jakości aplikacji komputerowych. Najbardziej znanym produktem tej firmy jest środowisko Rational Enterprise. Pozwala ono na przygotowaniu projektu zapisanego w języku UML. Następnie po odpowiednim przygotowaniu można wygenerować szkielet kodu w dowolnym języku przy pomocy środowiska programistycznego Microsoft .NET.

Jednym z narzędzi jest Rational PurifyPlus, które jest narzędziem testującym całego pakietu. Jest ono dostępne na platformę Linux. Jest to uniwersalna aplikacja pozwalające na badanie kodu aplikacji w języku C, C++ lub Java. Dzięki niej można:

- analizować wykorzystanie pamięci i wykrywać błędy programu dotyczące jej zarządzania („memory leaks") - błędy tego rodzaju są trudne do wykrycia

i jednocześnie jednymi z ważniejszych. Objawy problemów z pamięcią są nieprzewidywalne. Często powodują błędną pracę lub wręcz jej brak.

- badać wydajność w czasie rzeczywistym - narzędzia zawarte w aplikacji pozwalają na zgromadzenie danych dotyczących wydajności każdego elementu aplikacji. Ponadto wyniki pracy są przedstawiane w czytelny sposób, dzięki temu można wykorzystać tą wiedzę w celu poprawienia nieefektywnych części aplikacji.
- przeprowadzać analizę efektywności kodu - polega ona na sprawdzeniu, które części kodu są wykonywane i jak często. W efekcie jest to najlepsza droga do sprawdzenia wykorzystania kodu w trakcie działania aplikacji.
- tworzyć grafy zależności pomiędzy poszczególnymi funkcjami aplikacji - podczas wykonywania programu możliwe jest badanie zależności pomiędzy wszystkimi częściami kodu (na poziomie funkcji lub linii).

Biorąc pod uwagę możliwości, jakie daje „Rational PurifyPlus for Linux" testy zostaną wykonane za pomocą tego narzędzia. Dzięki temu zachowam jednakowe warunki przy analizie działania solverów SAT.

6.2 Przygotowanie danych wejściowych

Formuła, którą operuje solver SAT, jest zapisana w pliku wejściowym. Istnieją rozwiązania narzucające konieczność wpisywania badanej formuły za pomocą klawiatury. Jest to jednak uciążliwe i nieefektywne w przypadku dużych przykładów. Dlatego też większość solverów wymaga podania formuły w formie pliku z danymi.

Biorąc pod uwagę liczbę możliwych sposobów zapisania formuły logicznej został zaproponowany pewien ogólny standard zapisu formuły. Projektodawcą był DIMACS czyli „Center for Discrete Mathematics & Theoretical Computer Science". Jest to wspólny projekt takich uniwersytetów jak Princeton University czy Rutgers oraz laboratoriów badawczych min. AT&T Labs, Bell Labs, NEC, IBM Research lub Microsoft Research. Głównym zadaniem DIMACS jest rozwój i wsparcie prac z zakresu matematyki dyskretnej i teoretycznych nauk informatycznych. W efekcie tych działań są proponowane określone formaty danych, aby móc porównywać różne narzędzia analityczne i publikować zestawy problemów w postaci formuły logicznej.

Istnieją dwa formaty kodowania zaproponowane przez DIMACS. Pierwszy z nich, nazwany SAT, jest sposobem dla tych formuł logicznych, których sprowadzenie do innej postaci, przykładowo dysjunktywnej postaci normalnej, spowoduje zwiększenie jej długości. Ten typ danych nie jest używany w przypadku solverów SAT, ponieważ większość przykładów jest zapisana w konjunktywnej postaci normalnej (conjunctive normal form), stąd nazwa tego formatu CNF. Konjunktywna postać normalna to koniunkcja klauzul, które zawierają alternatywę literałów bądź ich negacji. Ta postać pozwala na przyśpieszenie pracy solverów, ponieważ aby cała formuła była spełniona, to każda klauzula powinna być spełniona. Natomiast dana klauzula będzie spełniona, jeśli dowolny literał z danej klauzuli przyjmie wartość 1. Jest to oczywiste ułatwienie pracy solverów SAT.

Plik w formacie CNF jest plikiem ASCII i składa się z dwóch części: preambuły (preamble) i klauzul (clauses). Preambuła zawiera informacje dla użytkownika i jest rozpoznawana poprzez znak 'c', który ją rozpoczyna. Po części komentarzy następuje linia określająca problem. Zaczyna się od znaku 'p' i zawiera następujące dane :

p FORMAT VARIABLES CLAUSES

Pole FORMAT wskazuje na format zapisu klauzul i dla formatu CNF powinno zawierać słowo 'cnf'. Pole VARIABLES zawiera liczbę w formacie integer, która oznacza liczbę zmiennych w przykładowej formule. Analogicznie, pole CLAUSES zawiera liczbę klauzul w danej formule.

Część zawierająca klauzule następuje po linii 'p'. Literały w formule są numerowane od 1 do liczby z pola VARIABLES. Nie jest konieczne, aby wszystkie zmienne zostały wykorzystane. Kolejne literały są oddzielane znakiem spacji lub tabulacji. Literały zanegowane są przedstawiane w postaci „-literał". Koniec klauzuli jest oznaczany cyfrą '0'. [1]

Format CNF jest używany przez większość solverów SAT. Dlatego też wybrane dane wejściowe, które poniżej opiszę, zostały zapisane w tym formacie.

Spośród wielu możliwych przykładowych formuł, które można uzyskać z różnych źródeł (Internet) wybrałem trzy, dzięki którym będę mógł przedstawić działanie wszystkich wybranych trzech solverów. Publikowane formuły są najczęściej dwóch typów. Pierwszy z nich to formuły generowane przy pomocy specjalnych programów całkowicie losowo. Drugi z nich to formuły tworzone na podstawie rzeczywistych

problemów z zakresu przykładowo matematyki dyskretnej. Poniżej prezentuję trzy formuły, których użyję jako danych wejściowych. Zostały one opublikowanie na stronie internetowej www.satlib.org.

a) pierwsza formuła została wygenerowana przez generator formuł logicznych zaprojektowany przez zespół Asahiro-Iwama-Miyano Formuła zawiera 100 zmiennych i 600 klauzul. Każda klauzula zawiera dokładnie 3 literały. Stosunek klauzul do literałów (ratio) wynosi 6. Ponadto jest spełnialna przez dokładnie jedno wartościowanie.

b) druga formuła dotyczy problemu kolorowania grafów. Polega on na decyzji, czy w danym grafie nieskierowanym istnieje określone kolorowanie tj. czy istnieje takie przyporządkowanie kolorów, że każde dwa sąsiednie wierzchołki mają różne kolory. Kodowanie grafu w formułę logiczną polega na przyporządkowniu do każdej zmiennej innego koloru wierzchołka oraz reprezentowaniu każdej krawędzi przez zbiór klauzul zapewniających różne kolory pomiędzy wierzchołkami. Ponadto są wprowadzone dodatkowe klauzule, aby zapewnić przypisanie dokładnie jednego koloru do wierzchołka. Przykładowa formuła zawiera 600 zmiennych i 2237 klauzul. Klauzule zawierają od 2 do 3 literałów i opisują graf o 200 wierzchołkach i 479 krawędziach.

c) ostatnia formuła opisuje problem analizy błędów w obwodzie elektronicznym (circuit fault analysis). Program sprawdzający obwody elektroniczne generuje serię przypadkowych, ale powtarzalnych symulacji procesu dla danego obwodu w celu znalezienia potencjalnych błędów. Następnie jest przygotowana z powstałych informacji formuła w formacie CNF. Przykładowa zawiera 1363 zmiennych i 3032 klauzul, których długość waha się od 1 do 6 literałów.

Poniżej przedstawiam tabelę zawierająca podstawowe informacje.

nazwa formuły	liczba zmiennych	liczba klauzul	ratio	SAT
aim-100-6_0-yes1-4.cnf	100	600	6	Tak
flat200-90.cnf	600	2237	3,73	Tak
ssa7552-159.cnf	1363	3032	2,22	Tak

Tabela 3 Podstawowe informacje o testowanych formułach

Jak można zauważyć przykładowe formuły są spełnialne. Pominąłem przykłady niespełnialne, aby móc swobodnie testować solvery SAT oparte na algorytmie SLS.

6.3 Solver zChaff

Poniżej przedstawiam statystyki, które są wyliczane przez solver w czasie jego pracy. MDL oznacza maksymalny poziom decyzyjny (DL - decision level), jaki został wybrany podczas pracy aplikacji. Pojęcia decyzji i implikacji zostały przedstawione powyżej. Czas pracy określa czas pracy aplikacji.

nazwa formuły	MDL	liczba decyzji	liczba implikacji	czas pracy
aim-100-6_0-yes1-4.cnf	7	15	265	0,13
flat200-90.cnf	39	172	13074	3,26
ssa7552-159.cnf	112	113	1365	0,35
średnia	53	100	4901	1

Tabela 4 Statystyki dotyczące pracy solvera

nazwa formuły	ilość PLK	ilość MLK	bloki PLK	bloki MLK	bloki w użyciu	razem[blok]	razem [bajt]
aim-100-6_0-yes1-4.cnf	4	7	7	102	13	122	188292
flat200-90.cnf	5	9	12	604	25	641	347884
ssa7552-159.cnf	4	9	19	1373	14	1406	414812
średnia	4	8	13	693	17	723	316996

Tabela 5 Statystyki dotyczące alokacji pamięci

Powyżej przedstawiam statystyki dotyczące alokacji pamięci. Zostały one utworzone na podstawie wyników pracy programu Rational Purify. PLK (potential leak) oznacza te miejsca w badanej aplikacji, które mogą być przyczyną błędów w przyszłości. MLK (memory leak) określa miejsce, gdzie doszło do błędów przy zarządzaniu pamięcią. Aplikacja Rational Purify pozwala na precyzyjne określenie występowania błędów pamięci, ponieważ podaje miejsce w kodzie źródłowym i krótkie uzasadnienie. Ponadto pozwala na określenie wielkości błędu poprzez podawanie wartości pamięci w jednostce liczby bajtów lub liczby bloków. Ostatni wiersz oznacza średnia arytmetyczna kolejnych kolumn.

Narzędzie Rational PureCoverage, jak już zostało to wspomniane wcześniej, służy do sprawdzenia wykorzystania kodu. Na podstawie testów przeprowadzonych przez to narzędzie, należy stwierdzić, że pod tym aspektem aplikacja zChaff została zbudowana niewydajnie. Wykorzystanie kodu źródłowego waha się od 35 do 45 %.

Jednak biorąc pod uwagę wszystkie wyniki, należy przyjąć założenie, że nie wszystkie funkcje zostały wykorzystane.

Poniżej przedstawiam wyniki działania ostatniego narzędzia Rational Quantify. Dzięki tej aplikacji można uzyskać informacje o czasie trwania danej aplikacji. Jednostką może być tutaj sekunda lub cykl maszynowy. Należy zaznaczyć, że testy były prowadzone na maszynie Pentium III z zegarem 1,16 Ghz. Kolumny oznaczone „funkcje aplikacji" i „funkcje systemowe" zawierają informacje o czasie trwania aplikacji tj. odpowiednio części wykonywanej w aplikacji i wykonywanej w postaci funkcji systemowych. Ostatnia kolumna przedstawia stosunek pomiędzy wartością z kolumny „funkcje aplikacji" a całkowitym czasem wykorzystanym przez tą aplikacją.

nazwa formuły	funkcje aplikacji [cykl]	funkcje systemowe [cykl]	ratio
aim-100-6_0-yes1-4.cnf	8705459	68144076	0,11
flat200-90.cnf	59540489	64938924	0,48
ssa7552-159.cnf	36133111	74026512	0,33
średnia	34793020	69036504	0,34

Tabela 6 Statystyki dotyczące czasu pracy

6.4 Solver march_eq

Poniżej przedstawiam statystyki, które podał solver w wyniku swojej pracy. Kolumna lookAhead oznacza liczbę wywołań funkcji o tej samej nazwie. Podobnie jest z następną kolumną.

nazwa formuły	lookAhead	unitResolve	czas pracy
aim-100-6_0-yes1-4.cnf	400	186	0,47
flat200-90.cnf	48326	13346	16,17
ssa7552-159.cnf	17702	792	0,1
średnia	22143	4775	6

Tabela 7 Statystyki dotyczące pracy solvera

nazwa formuły	bloki PLK	bloki MLK	bloki w użyciu	razem[bloki]	razem [bajty]
aim-100-6_0-yes1-4.cnf	1	2477	444	2922	247859
flat200-90.cnf	1	16466	2444	18911	362987
ssa7552-159.cnf	1	3432	506	3922	348541

Tabela 8 Wyniki dotyczące alokacji pamięci

Powyżej zamieszczam tabelę, w której zostały zamieszczone informacje

dotyczące alokacji pamięci. W porównaniu z rozdziałem 6.3 nie zostały zamieszczone dane o liczbie błędów pamięci (memory leaks - MLK) i potencjalnych błędów pamięci (PLK). Jest to związane z dużą ilością tych błędów oraz z pojawieniem się nowych rodzajów błędów pamięci. Otóż w trakcie pracy został wykryty kilkanaście razy błąd oznaczony ABR (Array Bounds Read). Zgodnie z dokumentacją programu Rational ten błąd oznacza próbę czytania przed lub poza adresami pamięci przydzielonej statycznie. Ponadto doszło do wystąpienia kilkunastu błędów typu MLK, o czym można przekonać się z powyższej tabeli.

Testy przeprowadzone przy pomocy narzędzia Rational PureCoverage pozwalają określić całkowite wykorzystanie kodu od 55% do 67%. Podobnie jak powyżej, nie wszystkie możliwe funkcje są wykorzystywane.

nazwa formuły	funkcje aplikacji [cykle]	funkcje systemowe [cykle]	ratio
aim-100-6_0-yes1-4.cnf	6234282	79210332	0,07
flat200-90.cnf	187589480	90568980	0,67
ssa7552-159.cnf	55821244	102132972	0,35

Tabela 9 Dane dotyczące czasu pracy

Powyżej przedstawiam wyniki działania ostatniego narzędzia Rational Quantify. Dzięki tej aplikacji można uzyskać informacje o czasie trwania danej aplikacji. Jednostką może być tutaj sekunda lub cykl maszynowy. Należy zaznaczyć, że testy były prowadzone na maszynie Pentium III z zegarem 1,16 Ghz.

6.5 Solver UnitWalk

Poniżej zostały przedstawione statystyki, które zostały przedstawione przez pracującą aplikację. Kolumna Assigns określa liczbę przyporządkowań, natomiast Flips to liczba zmian wartości literału. Szerzej te pojęcia zostały opisane powyżej.

nazwa formuły	Assigns	Flips	czas pracy
aim-100-6_0-yes1-4.cnf	401	101	0.4
flat200-90.cnf	3295200	741441	164,9
ssa7552-159.cnf	12267	813	0.5
średnia	1102623	247452	55

Tabela 10 Statystyki dotyczące pracy solvera

nazwa formuły	bloki PLK	bloki MLK	bloki w użyciu	razem[bloki]	razem [bajty]
aim-100-6_0-yes1-4.cnf	0	2	2	4	72
flat200-90.cnf	0	2	2	4	72
ssa7552-159.cnf	0	2	2	4	72
średnia	0	2	2	4	72

Tabela 11 Wyniki dotyczące alokacji pamięci

Dane, które zostały zamieszczone powyżej, pozwalają sądzić, że jest to najlepszy z solverów pod względem ilości pamięci, która została błędnie alokowana. Jednocześnie należy zwrócić uwagę, że cała aplikacja nie wymaga zbyt dużo pamięci.

Informacje, które zostały zebrane w trakcie pracy narzędzia Rational PureCoverage nakazują sądzić, że w trakcie przeprowadzanych testów około 69 % kodu zostało wykorzystane. Jest to wynik tego, że część kodu jest blokowana i wykorzystywana tylko w trakcie szukania ewentualnych błędów (debugging).

nazwa formuły	funkcje aplikacji [cykle]	funkcje systemowe [cykle]	ratio
aim-100-6_0-yes1-4.cnf	2443963	28648836	0,08
flat200-90.cnf	1847845092	2720312496	0,4
ssa7552-159.cnf	15673611	24901308	0,39
średnia	621987555	924620880	0,4

Tabela 12 Dane dotyczące czasu pracy aplikacji

Powyżej zamieszczam dane, które są wynikiem działania aplikacji Rational Quantify. Należy podkreślić, że czas, który tutaj jest podany w jednostkach cykl, można również przedstawić w sekundach. Niemniej jednak ta jednostka jest znacznie bardziej dokładna. Ostatnia kolumna przedstawia stosunek czasu, który aplikacja zużywa w swoich, wywoływanych funkcjach do całego czasu, który został wykorzystany zarówno przez program jak i wywołane funkcje systemowe.

6.6 Podsumowanie

Powyżej zostały zamieszczone wyniki testów przygotowanych przy pomocy narzędzi firmy Rational. Biorąc pod uwagę duże różnice pomiędzy wynikami w tej samej kategorii, przykładowo czasu pracy, dla tego samego solvera przy różnych danych wejściowych należy dokonać podsumowania uwzględniając ten fakt. Dlatego też poniżej porównuję wyniki dla różnych solverów przy tych samych formułach wejściowych.

nazwa formuły	zChaff	march_eq	UnitWalk
aim-100-6_0-yes1-4.cnf	0,13	0,47	0.4
flat200-90.cnf	3,26	16,17	164,9
ssa7552-159.cnf	0,35	0,1	0.5

Tabela 13 Porównanie czasu pracy

Powyższa tabela przedstawia porównannie czasu pracy, które zostało obliczone przez działający solver i przedstawione jako jedna z jego statystyk. Jak można zauważyć, zChaff najlepiej sobie radzi z dwiema pierwszymi instancjami. Trzecia formuła jest najszybciej obliczana przez solver march_eq. Natomiast UnitWalk działa w stosunku do pozostałych wolno. Przypadek pierwszy wydaje się być wyjątkiem.

nazwa formuły	zChaff	march_eq	UnitWalk
aim-100-6_0-yes1-4.cnf	122	2922	4
flat200-90.cnf	641	18911	4
ssa7552-159.cnf	1406	3922	4

Tabela 14 Porównanie liczby bloków

Powyższa tabela przedstawia wykorzystanie pamięci przez każdy solver. Precyzyjnie, jest to liczba bloków alokowanych przez daną aplikację zarówno tych błędnie przypisanych („memory leak") jak i tych, które były później wykorzystane. Pod tym kątem najlepszym solverem jest UnitWalk. Jednak ten wynik jest prawdopodobnie efektem typu algorytmu, a nie jakości kodu. Jeśli dokonamy porównania pod kątem tego samego algorytmu czyli rozpatrzymy wypadki zChaff i march_eq, to można wyprowadzić wniosek, że zChaff jest lepszym solverem. Ponadto należy przypomnieć, że dla march_eq zostały znalezione błędy dotyczące odwołań do adresów spoza tablic (ABR). Takich błędów nie wykryto dla zChaffa.

solver	zChaff	march_eq	UnitWalk
wykorzystanie kodu [%]	35-45	55-67	69

Tabela 15 Porównanie stopnia wykorzystania kodu

Powyższa tabela zawiera porównanie stopnia wykorzystania kodu na podstawie danych przedstawionych przez narzędzie Rational PureCoverage. Jak można się przekonać najwydajniej jest wykorzystany kod dla solvera UnitWalk. Jednak w tym przypadku nie należy wyciągać pochopnych wniosków. Dane te dotyczą jedynie stopnia

wykorzystania funkcji. Może się zdarzyć, że dane wejściowe są na tyle proste, że nie wymuszają wykorzystania pewnych określonych partii kodu. W ten sposób wyniki nie odzwierciedla prawidłowych zależności pomiędzy tymi solverami.

nazwa formuły	zchaff	march_eq	unitwalk
współczynnik ratio	0,38	0,5	0,4

Tabela 16 Porównanie współczynnika ratio

Powyższa tabela dotyczy współczynnika ratio. W trakcie badań aplikacji narzędziem Rational Quantify zostały przeprowadzony pomiar czasu trwania aplikacji z podziałem na czas wykorzystany w aplikacji i czas wykorzystany w wywołaniach funkcji systemowych czyli poza aplikacją. Współczynnik ratio przedstawia stosunek czasu zużytego przez funkcje aplikacji do całego czasu wykorzystanego przez aplikację. Pod tym aspektem wszystkie trzy solvery niezależnie od dancyh wejściowych mają podobne wyniki. W zależności od solvera współczynnik ratio waha się od około 0,38 do około 0,5. Wydaje się to być całkiem dobry wynik, ponieważ wskazuje, że większość czasu jest spędzana na pracy na poziomie aplikacja - system operacyjny. Jest to korzystna informacja, ponieważ ta część środowiska aplikacji zależy od systemu operacyjnego, a szczególnie sposobu i jakości jego podstawowych funkcji.

Z przedstawionych powyżej statystyk wynika, że solverem, który najlepiej rozwiązywał postawione zadania, był zChaff. Należy w tym miejscu zauważyć, że celem powyższych działań tj. wykonanych testów nie było przedstawienie najlepszego solvera. Takie działania są wykonywane w trakcie konkursu „SAT Contest". Z całą pewnością można powiedzieć, że zChaff jest przykładem dobrze zaprojektowanego i dobrze zbudowanego solvera. Wskazują za tym wyniki, jakie przedstawiłem powyżej.

Ostatnią rzeczą o jakiej trzeba wspomnieć jest to, że powyżej przedstawiłem część wyników. Duża część wyników okazała się nieprzydatna do moich celów. Zestaw narzędzi firmy Rational pozwala na precyzyjne określenie błędów w projektowanej aplikacji oraz na zwiększenie jej jakości. Przykładowo, Rational PureCoverage oprócz określenia stopnia wykorzystania kodu dla całego projektu wygenerowało również informacje o wykorzystaniu poszczególnych fragmentów kodu. Zgodnie z dokumentacją można przeprowadzić taki test na poziomie poszczególnych linii badanego kodu.

7 Wnioski

W przedstawionej pracy udało mi się przedstawić algorytmy, które efektywnie rozwiązują problem SAT. Podobnie opisałem obecny stan badań nad tą częścią teorii złożoności obliczeniowej poprzez zanalizowanie najnowszych solverów SAT.

Informacje, które zebrałem podczas przygotowywania tej pracy, pozwalają wnioskować, że :

- problem SAT i próby jego efektywnego rozwiązania są silnie zaawansowane;
- solvery SAT nie stanowią przełomu w teorii algorytmów, natomiast są niezwykle skutecznymi aplikacjami;
- kierunek prowadzonych badań zmierza ku optymalizacji solverów SAT;

Literatura

[1] "Satisfiability Suggested Format",
[2] Aho A.V. , Hopcroft J. , Ullman J.D. "Projektowanie i analiza algorytmów komputerowych", Warszawa, PWN, 1983
[3] Błażewicz J. "Złożoność obliczeniowa problemów kombinatorycznych", Warszawa, WNT, 1988
[4] Cook S. "The P versus NP Problem", http://www.claymath.org/millennium
[5] Cormen T.H. ,Leiserson C.,Rivest R. "Wprowadzenie do algorytmów", Warszawa, WNT, 2003
[6] Kautz H., Selman B., McAllester D. "Walksat in 2004 Competition", www.cs.washington.edu/homes/kautz/papers/
[7] Malik S. "The Quest for Efficient Boolean Satisfiability Solvers", www.princeton.edu/~chaff/publication/
[8] Moskewicz M., Madigan C., Zhao Y., Zhang H., Malik S. "Chaff: Engineering an Efficient SAT Solver ", research.microsoft.com/ users/lintaoz/papers
[9] Papadimitriou CH. H. "Złożoność obliczeniowa,, , Warszawa, WNT, 2002
[10] Rumbaugh J., Jacobson I., Booch G. "The Unified Modeling Language Reference Manual, Second Edition", Addison Wesley, 2005
[11] Silva J. "The impact of branching heuristics in propositional satisfiability algorithms,,, http://sat.inesc.pt/~jpms/research
[12] Simon L., Le Berre D. "SAT04 Contest", www.lri.fr/~simon/contest/ results/
[13] Stutzle T. "Stochastic Local Search", www.intellektik.informatik.tu-darmstadt.de/~tom/Lehre/WS03-04/SLS/
[14] Zhang H., Stickel M.E. "Implementing the Davis-Putnam Method", ftp://ftp.cs.uiowa.edu/pub/hzhang/sato/papers/
[15] Zhang L., Madigan C.F., Moskewicz M.H., Malik S. "Efficient conflict driven learning in a Boolean Satisfiablity Solver", www.princeton.edu/~chaff/ publication/